Transfer Control for Resilient End-to-End Transport

By

Truc Anh N. Nguyen

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science

Thesis Committee:

Dr. James P.G. Sterbenz: Chairperson

Dr. Victor S. Frost

Dr. Gary J. Minden

Date Defended: June 05, 2014

The Thesis Committee for Truc Anh N. Nguyen certifies
that this is the approved version of the following thesis:

Transfer Control for Resilient End-to-End Transport

Committee:

Dr. James P.G. Sterbenz: Chairperson

Date Approved: September 09, 2014

Abstract

Residing between the network layer and the application layer, the transport layer exchanges application data using the services provided by the network. Given the unreliable nature of the underlying network, reliable data transfer has become one of the key requirements for those transport-layer protocols such as TCP. Studying the various mechanisms developed for TCP to increase the correctness of data transmission while fully utilizing the network's bandwidth provides us a strong background for our study and development of our own resilient end-to-end transport protocol. Given this motivation, in this thesis, we study the different TCP's error control and congestion control techniques by simulating them under different network scenarios using ns-3. For error control, we narrow our research to acknowledgement methods such as cumulative ACK - the traditional TCP's way of ACKing, SACK, NAK, and SNACK. The congestion control analysis covers some TCP variants including Tahoe, Reno, NewReno, Vegas, Westwood, Westwood+, and TCP SACK.

I like to dedicate this work to my parents for their continuous support and guidance.

Acknowledgements

I would like to thank to my comittee members, especially my advisor Dr. James P.G. Sterbenz for his support and guidance. I would also like to thank all ResiliNets group members for their help and kindness. Finally, many thanks to my family and friends, especially my parents for always supporting me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Residing between the network layer and the application layer, the transport layer transfers data between two communicating processes running on two end systems. The transport layer is an end-to-end analog of the hop-by-hop network layer's data delivery service [5]. The services provided by the transport layer enable application-layer processes to exchange information without having to worry about the underlying network's architecture. It is important to distinguish between a transport service and a transport protocol. While a transport service refers to a set of functions the transport layer offers to the application layer, a transport protocol specifies a set of rules that a pair of transport sender and receiver follows while cooperating with each other to provide a particular service [6].

Given the unreliable nature of the network layer, in order to provide reliable data transfer–correct and in-order data delivery with no loss and no duplications, a reliable transport-layer protocol such as the Internet's Transmission Control Protocol (TCP) [7] has to incorporate various mechanisms. The different features of a reliable transport protocol include connection management, error control, flow control, and congestion control. Connection management specifies how a

pair of sender and receiver synchronize their establishment and termination of a connection to prevent data loss and duplication mainly caused by the confusion between different connections. Error control consists of a set of algorithms to detect and recover from data loss and corruption. Flow control is the technique employed to prevent the sender from overflowing the receiver's buffer. Finally, congestion control ensures an appropriate sending rate to avoid overwhelming the network while still being able to achieve reasonable bandwidth utilization. Error control, flow control, and congestion control are referred as transport-layer transfer control.

1.1 Problem Statement

The development of any new resilient end-to-end transport protocol requires a thorough understanding of the existing reliable protocols, especially TCP, its different features with the main focus on transfer control, and its proposed enhancements, variants, and extensions.

1.2 Proposed Solution

Using the open source network simulator ns-3 [8], we study transport-layer transfer control mechanisms by simulating them under different network scenarios. For error control, we focus on the various acknowledgement techniques including the traditional positive acknowledgment (ACK), selective acknowledgment (SACK), negative acknowledgment (NAK), and a hybrid version of both SACK and NAK called selective-negative acknowledgment (SNACK). For congestion control, we study the different mechanisms employed in the well-known

TCP variants: Tahoe, Reno, NewReno, Vegas, Westwood, Westwood+, and TCP SACK. We analyze their behaviors, weaknesses and strengths, and suggest possible ways to enhance them based on our analysis. The knowledge gained from this study will contribute to the design of our resilient end-to-end transport protocol, Res-TP whose initial development has been conducted on by other members in the group [9, 10].

TCP plays a central role in our analysis [6]; since its inception, TCP has been the dominant reliable transport-layer protocol developed for the Internet. The wide deployment of TCP has exposed it to many different issues that trigger extensive study resulting in many enhancements and extensions. Hence, studying TCP and its variants gives us significant insight to the evolvement of transport-layer design. Finally, documents on TCP development are publicly available.

1.3 Contributions

The contributions of this thesis are listed below:

- implement the conservative SACK-based loss recovery algorithm for TCP [11] in ns-3
- modify the above SACK-based algorithm to incorporate NAK and SNACK in replacement of SACK to study the different acknowledgment techniques
- collaborate with other group members to implement TCP Westwood+
- implement TCP Vegas congestion control mechanism in ns-3
- implement a Burst Error Model in ns-3 based on the existing Rate Error Model

- implement SCPS-TP and its essence relevant for this thesis is SNACK and TCP Vegas
- analyze the performance of the various acknowledgment and congestion control mechanisms under different network environments

1.4 Organization

The organization of the thesis is as follows: Section 2 gives a general overview on the error and congestion control features of a reliable transport layer protocol followed by a discussion on the specific mechanisms/protocols simulated and studied in the thesis. Section 3 provides some details on how the transport layer module and its different classes are organized and interact in ns-3. In this context, we show how we implement all the mechanisms needed to accomplish our research goal. In the next section, Section 4, we present our simulation model followed by the results represented in terms of plots on different performance metrics and our analysis on those results. We conclude the thesis in Section 5 in which we highlight what we have learned from our research together with some directions for future work.

Chapter 2

Background and Related Work

In this section, we explain the four key acknowledgment schemes ACK, SACK, NAK, and SNACK together with the different congestion control mechanisms employed in TCP Tahoe, Reno, NewReno, Vegas, Westwood, Westwood+, and TCP SACK. To be comprehensive, we start our discussion with a timeline on the evolution of TCP (Figure 2.1) and an overview on the early TCP implementations. The section ends with a brief survey on other related work.

2.1 Early TCP Implementations

Early TCP implementations [7] used the Go-Back-N Automatic Repeat Request (ARQ) mechanism (Figure 2.2(b)) together with positive ACK and a retransmission timer on the sender's side to recover from data loss. The Go-Back-N allows multiple TCP segments to be in flight simultaneously, which is different from the earlier Stop-and-Wait approach (Figure 2.2(a)). Associated with each segment is a retransmission timer set by the sender when the segment is sent. To prevent premature retransmission, the timer's value is supposed to be greater than

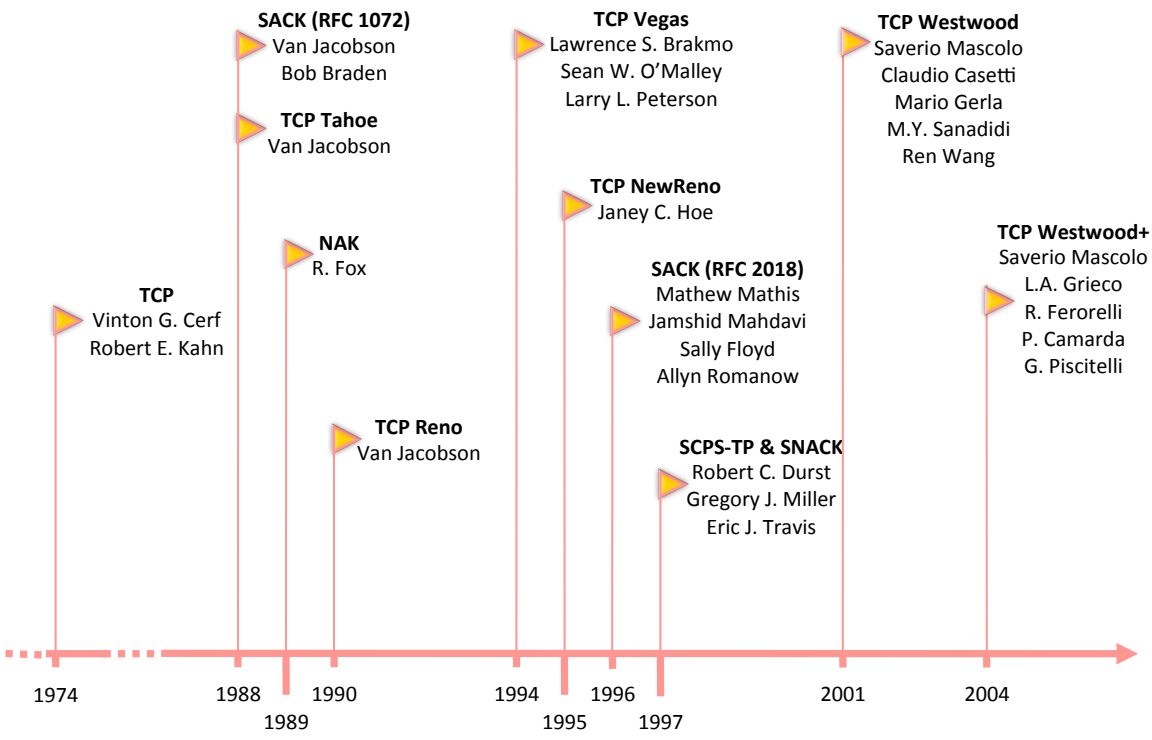

Figure 2.1. Evolution of TCP

the network's round-trip delay to allow sufficient time for the segment to reach the receiver and its acknowledgement to traverse back before the timer fires. On the other end, the receiver is expected to acknowledge every received segment. When an out-of-order segment arrives due to a lost or corrupted segment, the receiver retransmits the previous ACK. The sender views a duplicate ACK as an indication of data loss, but it does not retransmit the segment until its retransmission timer expires. This behavior differentiates the Go-Back-N mechanism from the Fast Retransmit (Figure 2.2(c)) explained in later section. Upon a timeout, the sender goes back and retransmits all the segments starting with the oldest loss based on the acknowledgment number in the received ACK. Early TCP implementations employed no congestion control techniques.

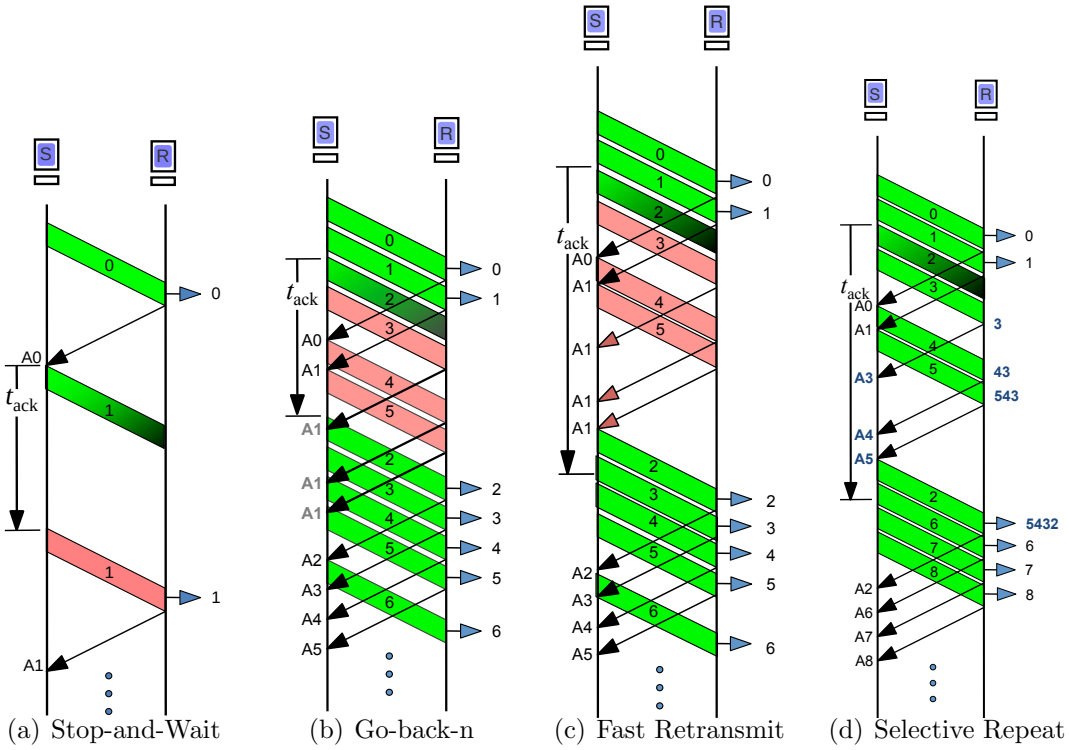

Figure 2.2. TCP retransmission mechanisms (adapted from [1])

2.2 Error Control

In order to detect and recover errors arising from packets being transferred through the underlying unreliable network, a reliable transport protocol uses multiple approaches including sequence numbers, checksums, acknowledgments, and retransmissions. A packet is marked as being errored if it is corrupted, misordered, or duplicated. In this thesis, we focus on the various acknowledgment mechanisms: positive ACK, selective ACK, negative ACK, and selective-negative ACK. Even though we do not explicitly study the retransmission schemes, our analysis on the TCP variants cover some of them. Fast Retransmit is part of Tahoe, Reno, and NewReno while TCP SACK combines selective ACK and selective repeat retransmission policy (Figure 2.2(d)).

2.2.1 ACK

TCP uses a cumulative acknowledgment (ACK) scheme in which the data receiver sends a positive ACK to acknowledge all data octets that are below the acknowledgment number. The acknowledgment number contained in the ACK packet's header indicates the receiver's next expected data byte. Any new, but unexpected data arriving at the receiver following a loss will be buffered, but not acknowledged. Instead, out-of-order data triggers a retransmission of the last ACK packet. TCP depends heavily on its ACK stream to decide when it should introduce new packets into the network in order to maintain its throughput.

Despite of its simplicity, TCP's cumulative acknowledgment mechanism has many drawbacks, especially when TCP being deployed in long fat networks (LFNs) – networks containing paths with high bandwidth $\times$ delay product. These networks require a large amount of unacknowledged data to fill the pipe that increases the probability of having multiple segment losses per sending window. The use of a single acknowledgment number in an ACK packet cannot give the sender a complete view of the receiver's buffer status. As mentioned earlier, the receiver buffers out-of-order data but does not acknowledge them. The lack of knowledge on the sender side may result in multiple unnecessary retransmissions that waste the network bandwidth and drop the throughput.

A TCP sender has to wait at least a full round trip time (RTT) until a duplicate ACK arrives to learn about each packet loss. In a long delay network environment, by the time the sender realizes a loss and resends the missing segment, the sending window may have already been exhausted. An empty window prevents the sender from sending new data until the receiver receives the retransmitted packet and sends a cumulative ACK back. In this case, after receiving the ACK, the sender

would have to refill the empty pipe due to the stall [3]. Furthermore, because of the limited information carried by an ACK, TCP can only recover at most one loss per window. The presence of multiple losses will cause TCP to loose its ACK-based clock and degrade its performance.

2.2.2 SACK

Selective Acknowledgement (SACK) is a TCP extension that addresses the cumulative ACK limitations in facing multiple segment losses. It was first proposed in [12] and revised in [2] to enhance its robustness. Unlike ACK with a single acknowledgement number, SACK carries a number of blocks specifying all the data segments that have been received but not acknowledged due to a gap in the receiver's buffer. With this information, the sender can avoid duplicate retransmissions. Furthermore, the sender is able to retransmit multiple segments at once without having to wait for the next ACK to arrive if the receiver window and the congestion window are not the constraints.

Figure 2.3. Sack-Permitted option [2]

The TCP SACK extension consists of two TCP options; one is the Sack-Permitted, and the other is the SACK option itself. The Sack-Permitted is a two-byte option sent in the SYN segment at the beginning of a connection to inform the receiver the sender's capability of processing SACK information (Figure 2.3). Only upon receiving the Sack-Permitted, the receiver is allowed to transmit SACK options when needed. The SACK option itself contains multiple SACK blocks in which each block represents a contiguous and isolated chunk of data after the first

gap in the receiver's buffer. As shown in Figure 2.4, each SACK block is defined by two 32-bit sequence numbers called left edge and right edge. The left edge denotes the first sequence number of the block while the right edge is one number after the last sequence number of the block. In other words, all sequence numbers or data bytes that are smaller than the left edge, and those that are equal or greater than the right edge are missing in the receiver's buffer.

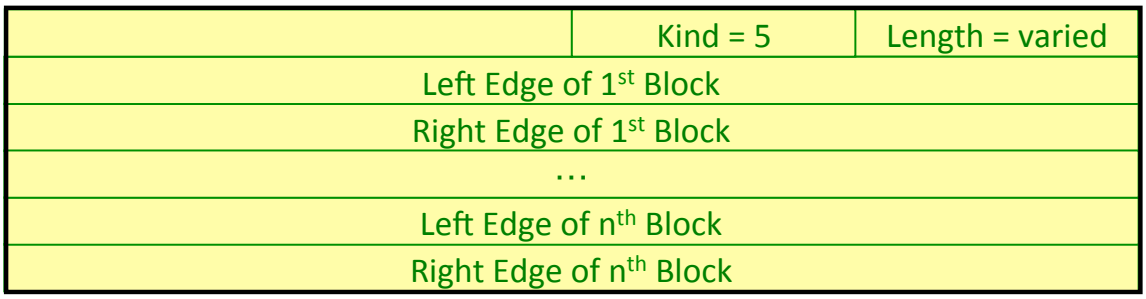

Figure 2.4. SACK option [2]

The order of all SACK blocks within a SACK option is critical for this TCP extension to achieve its full potential. The first SACK block has to cover the most recently received segment to reveal the current network and receiver's buffer status. The remaining SACK blocks should be constructed by replicating the most recently reported blocks first. The construction allows at least 3 notifications of a single isolated chunk of data, which is redundant but useful in a lossy ACK channel. In addition, when generating a SACK option, the receiver should include as many SACK blocks as needed and allowed. Given the limitation of 40 bytes for TCP options, with each SACK block occupies 8 bytes, each SACK option can carry a maximum number of 4 blocks. In the presence of other TCP options, this number will be reduced; there is typical with the TCP timestamp option using the fourth.

2.2.3 NAK

Negative acknowledgement (NAK) [3] is another extension proposed to enhance TCP's ACKing technique. Unlike SACK, NAK has never been under the standardization process and has recently been moved to historic status due to its lack of deployment [13]. Instead of acknowledging segments that have been received successfully like SACK, a NAK option informs the sender of a missing segment in the receiver's buffer. A NAK option is 7 bytes in length consisting of 4 fields. In addition to the two 1-byte type and length fields that are common in any TCP options, a NAK contains another 4-byte field specifying the first sequence number of the reported gap, and a 1-byte field specifying the size of the gap in segments. While a SACK block may be sent multiple times, a NAK is not repetitious to prevent unnecessarily NAKing and retransmitting only the first missing segment.

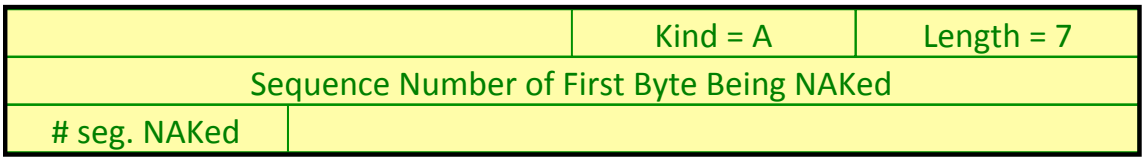

Figure 2.5. NAK option [3]

Below are a few remarks on SACK and NAK:

- Both SACK and NAK are sent unreliably. The redundancy in SACK block construction is the only means for recovering a lost SACK. However, a high number of SACK drops may still cause unnecessary data retransmissions.
- Both SACK and NAK are advisory information. The sender is not required to retransmit the missing segments upon a SACK or NAK receipt.

- By sending a SACK or a NAK whenever a gap exists in the buffer, the receiver assumes that the missing segment is lost and will never arrive unless being retransmitted. The assumption results in a waste of bandwidth in case the segment is delayed. As stated in [3], if this scenario is rare, the drawback is insignificant.

- In comparison between SACK and NAK, SACK is more complicated due to the complex state information that needs to be maintained for constructing SACK blocks, especially when the ACK delay mechanism is implemented. On the other hand, under a lossy environment in which the spacing between segment losses is close, the recovery process is much faster because many lost segments can be reported in a single SACK option [3].

We briefly discuss the shortcomings of SACK and NAK in space communication that motivated the development of SNACK, another acknowledgment scheme we study in this thesis:

- *The shortcomings of SACK*:

  Given the capacity-limited, error-prone, and asymmetric channels of a space communication network, the use of SACK poses two main issues: bit efficiency and information constraint [14]. The limited bandwidth may not be able to afford the use of 8 bytes for a single data chunk. Furthermore, in an environment with a high corruption level, 3 SACK blocks per SACK message may not be sufficient to give the sender a complete view of the receiver's buffer. This restriction prevents the sender from reacting promptly to the network's dynamics. The scenario becomes worse if the ACK channel is tuned to reduce the load on the small-capacity ACK link, which further reduces the amount of SACK packets arrived at the sender. When SACK

is used with the Fast Retransmit algorithm, the probability of receiving a certain number (usually 3) of duplicate ACKs on the bandwidth-constrained ACK channel to trigger a retransmission is also lower. Upon the expiration of the retransmission timeout, the existing SACK information held by the sender must be cleared and the sender loses its benefits from SACK. In this error-prone environment, ACK packets may also be lost.

- *The shortcomings of NAK*:

  Similar to SACK, NAK has its own disadvantages in space communications even though it is very bit-efficient. A single NAK message can only specify one hole in the receiver's buffer. In a lossy environment, it is desirable for the receiver to be able to notify the sender as many missing segments as possible. The effect is more significant when the ACK channel is tuned to reduce the ACK frequency. Space networks have long delay, which is another factor that limits the sender's capability to efficiently recover from multiple losses due to the very small amount of information it receives from the other end [14].

2.2.4 SNACK

Developed in the context of SCPS-TP – a TCP extension for space communications [4, 15], Selective Negative Acknowledgment (SNACK) is a hybrid version of both SACK and NAK. Similar to NAK, SNACK is a negative acknowledgment that acknowledges missing segments in a bit-efficient manner. Inheriting from SACK, a SNACK packet can inform multiple holes in the receiver's buffer. While SACK and NAK are advisory, SNACK is a request for a retransmission. When the sender receives a SNACK message, it must stop its data transmission and

retransmit the missing segments before resuming its normal operation. In this case, SNACK can operate without the need for Fast Retransmit.

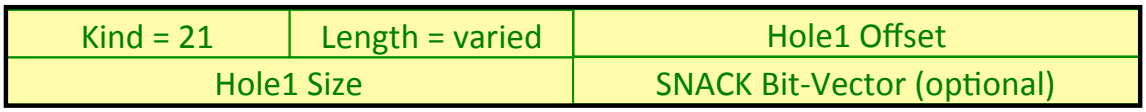

Figure 2.6. SNACK option [4]

A length-varied SNACK option consists of 4 mandatory fields and an optional field (Figure 2.6). Apart from the common type and length fields, the 2-byte offset field specifies the displacement from the ACK sequence number of the first reported hole, while the 2-byte size field specifies the hole's length in maximum segment size (MSS) unit. The last field in a SNACK option is an optional SNACK Bit-Vector that is used to report additional holes in the receiver's buffer. A 0 bit represents a missing segment, and a 1 bit represents a received segment.

2.2.5 Example of SACK, NAK, and SNACK

In this section, we show an example to illustrate how SACK, NAK, and SNACK options are generated. Part of the example is adapted from [2]. We assume that the sender transmits a burst of six segments with 1460 bytes of data in each to the receiver after establishing a connection through the three-way handshake. Also, we assume that the initial sequence number for the connection is 0. Due to the noisy channel, the second, the fourth, and the fifth segments are dropped (Figure 2.7).

When the third segment (segment 2) arrives, the receiver realizes a hole in its buffer. When sending an ACK that re-acknowledges the first segment (segment 0), the receiver attaches a SACK, a NAK, or a SNACK option whose structure is

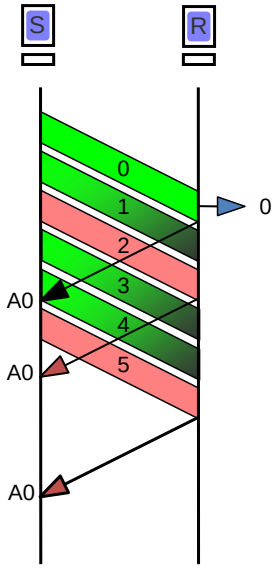

Figure 2.7. SACK, NAK, and SNACK Example

	Kind = 5	Length = 10
Left Edge of 1st Block = 2920		
Right Edge of 1st Block = 4380		

Figure 2.8. SACK option when segment 2 arrives

	Kind = A	Length = 7
Sequence Number of First Byte Being NAKed = 1460		
# seg. NAKed = 1		

Figure 2.9. NAK option when segment 2 arrives

Kind = 21	Length = 6	Hole1 Offset = 0
Hole1 Size = 1		

Figure 2.10. SNACK option when segment 2 arrives

demonstrated in Figures 2.8, 2.9, and 2.10.

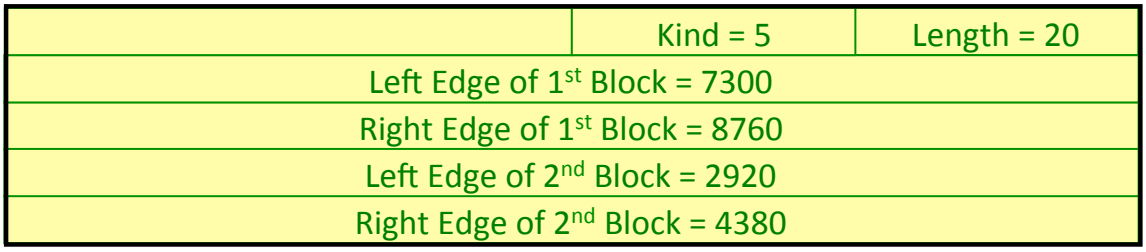

Figure 2.11. SACK option when segment 5 arrives

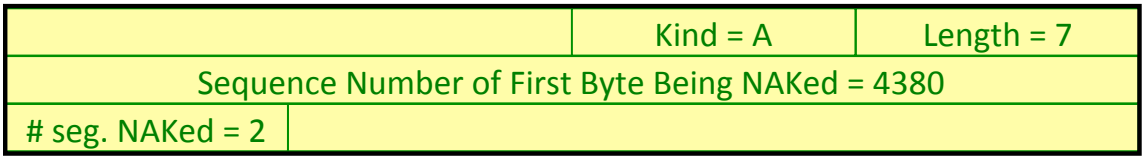

Figure 2.12. NAK option when segment 5 arrives

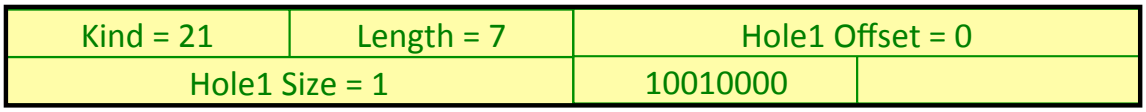

Figure 2.13. SNACK option when segment 5 arrives

When segment 5 arrives, due to another 2 segments being dropped during the transmission, the receiver sends another SACK, NAK, or SNACK option signifying the gaps in its buffer (Figures 2.11, 2.12, 2.13). To better illustrate the option generations, we assume that segment 1 is still missing by the time the receiver receives segment 5.

2.3 Congestion Control

Since its inception, congestion control has been under extensive studies in the research community due to its complexity, its importance, and the rapid growth of

the Internet. The expansion of the Internet to include networks with unique characteristics such as mobile wireless and space raise the need for a re-investigation of the existing mechanisms. Most of the time, a modification in the congestion control algorithm forms a new variant, resulting in the numerous TCP versions at the current time. As mentioned earlier, congestion control is a mechanism used by a data sender to adjust its sending rate according to the network's status. In this section, we explain the congestion control algorithms we study in the thesis.

Congestion control algorithms can be classified into 4 categories [16]:

1. Loss-based algorithms interpret packet loss as a signal of network congestion. The standard TCPs: Tahoe [17], Reno [18,19], NewReno [20,21] , SACK [11], Westwood [22], and Westwood+ [23], and some of the high-speed protocols: STCP [24], HSTCP [25], BIC [26], and CUBIC [27] fall into this class.

2. Delay-based algorithms (CARD [28], Vegas [29] , and FAST [30]) consider the increasing in delay due to the queue build up when load exceeds network capacity as an indication of congestion.

3. Hybrid algorithms employ both loss- and delay-based mechanisms. Africa [31], Compound [32], and YeAH [16] are a few examples of mixed loss-delay-based TCP variants.

4. Explicit congestion notification (ECN) relies on explicit signal from routers to learn about network congestion; XCP [33] is an example.

2.3.1 TCP Tahoe

TCP Tahoe [17] is the earliest variant that implements Van Jacobson's congestion control algorithm after the occurrence of a series of *congestion collapses*

in 1986 [17]. The algorithm introduces a new congestion window parameter to limit the sending rate and employs a principle of *packet conservation*. The sending rate is the minimum of the receive window and the congestion window to avoid overloading both the receiver and the network. A connection cannot run stably, or be *in equilibrium* unless it obeys the *conservation of packets* principle in which a new packet should not be placed into the network before previous packet has left. Following the principle, TCP Tahoe comprises several new algorithms to handle congested conditions, including:

- *Slow-start*: The algorithm allows a connection to get to equilibrium by gradually increasing the amount of sending data to fill the pipe when the connection starts or after a timeout.

- *Round-trip-time variance estimation*: The round trip time estimator conserves a connection's equilibrium after slow start by accurately estimating the sender's retransmit timer, which controls when the sender places a new packet into the network.

- *Congestion avoidance*: The algorithm is responsible for signaling the endpoints of a congestion event so that they can adjust utilization accordingly.

- *Fast retransmit*: The fast retransmit allows more timely loss recovery on the sender's side. Instead of waiting for the retransmission timer to fire, the algorithm uses the receipt of a certain number of duplicate acknowledgments (usually called dupACK threshold and set to 3) as a trigger for packet's resend.

Figure 2.14 illustrates Tahoe's congestion control state transition diagram. Starting with the slow-start phase after establishing the connection, the conges-

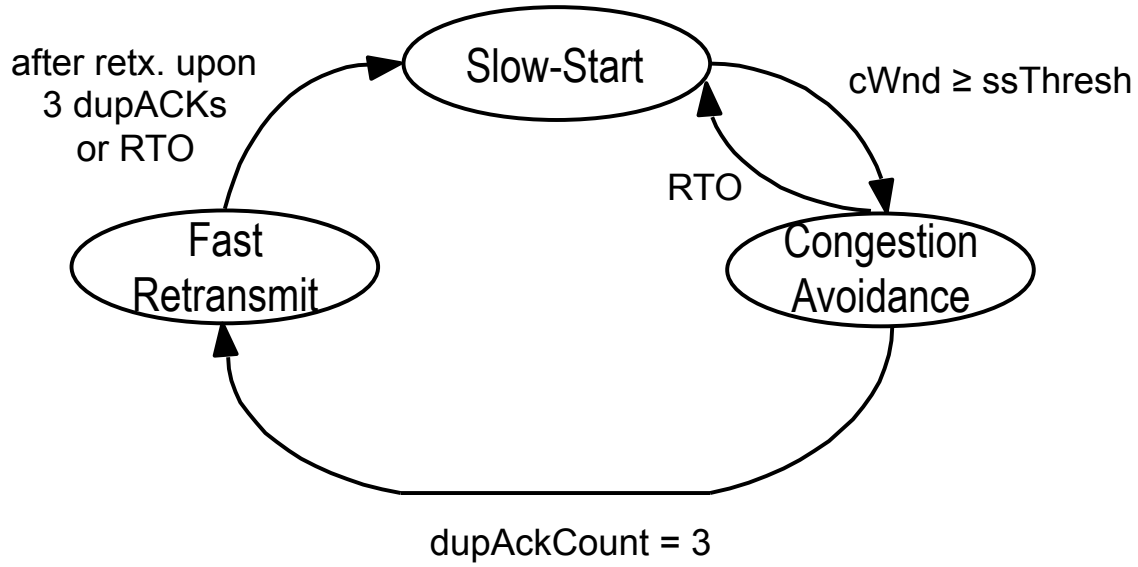

Figure 2.14. Tahoe congestion control state transition diagram

tion window is increased by one on the receipt of each ACK. This means that the congestion window is double every RTT resulting in an exponential increase of the sending rate. Tahoe remains in the slow start until the congestion window reaches the slow-start threshold upon which the sender moves to congestion avoidance. While in the congestion avoidance phase, the congestion window is increased by one every RTT resulting in a linear increase of the sending rate to prevent network's overwhelming. While in slow start or congestion avoidance, if the sender receives up to 3 duplicate ACKs, it transfers to the fast retransmit state, retransmits the missing segment, reduces the slow-start threshold to half of the current congestion window, resets the congestion window to its initial value of one, and switches back to slow start to re-fill the transmission pipe all over again. Whenever a retransmission timer (RTO) fires, Tahoe transfers to slow start by reseting the congestion window to 1 segment size.

2.3.2 TCP Reno

In addition to those algorithms proposed in TCP Tahoe, TCP Reno [18, 19] introduces a new mechanism called *Fast Recovery* with an attempt to enhance TCP performance in high bandwidth×delay product networks. After receiving 3 duplicate ACKs, retransmitting the lost packet, and reducing the slow start threshold to half of the current congestion window, instead of performing slow start as in Tahoe, TCP Reno goes through the fast recovery phase, transmitting a new segment on the receiving of each additional duplicate ACK if the congestion window and the receiver's advertised window values allow. The fast recovery phase ends when an ACK acknowledging new data arrives that causes the sender to switch back to congestion avoidance by setting the congestion window to the current slow start threshold instead of one as depicted in Figure 2.15. With this algorithm, the transmission pipe does not empty after a loss, and we do not have to waste time and resources re-filling it. Reno switches back to slow start only upon the expiration of an RTO.

2.3.3 TCP NewReno

TCP NewReno [20,21] consists of a slight modification to its predecessor Reno's fast recovery algorithm resulting in higher throughput. While TCP Reno improves the performance when there is a single packet loss per window, it functions poorly in the presence of multiple losses. To address the issue, instead of leaving the fast recovery state when receiving a partially new ACK as in Reno, TCP NewReno remains in the state until all of the data transmitted before it enters fast recovery is acknowledged. In other words, NewReno treats every partial ACK as an indication that the packet following the ACK has been lost and needs to be retransmitted.

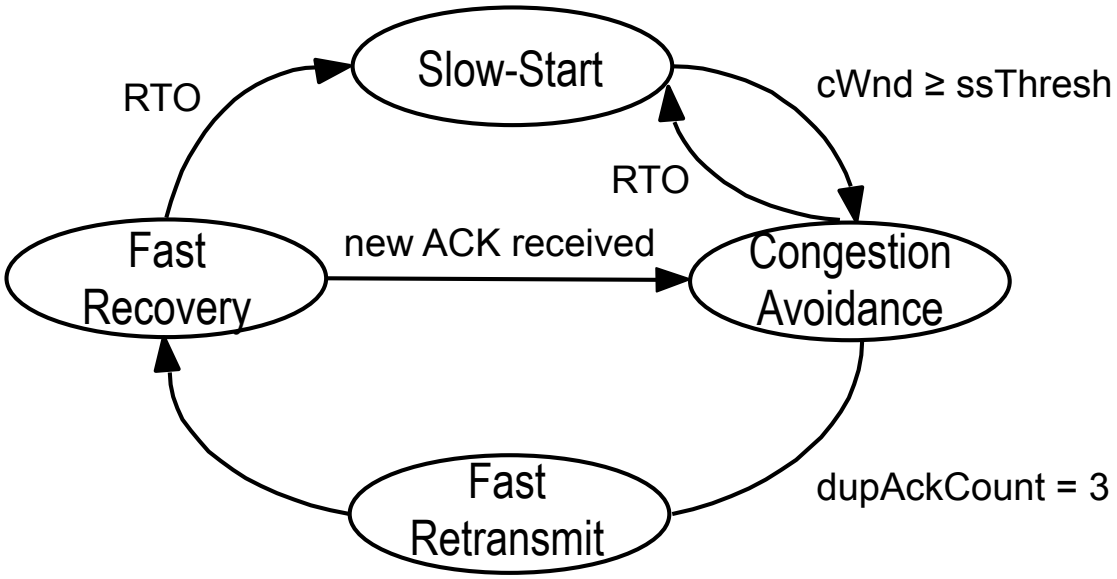

Figure 2.15. Reno congestion control state transition diagram

Hence, it can recover from multiple packet losses without having to wait for the retransmission timeout or to re-enter the fast recovery phase multiple times as in Reno. Figure 2.16 illustrates the transition between its congestion control states. NewReno transfers back to slow start only when an RTO fires.

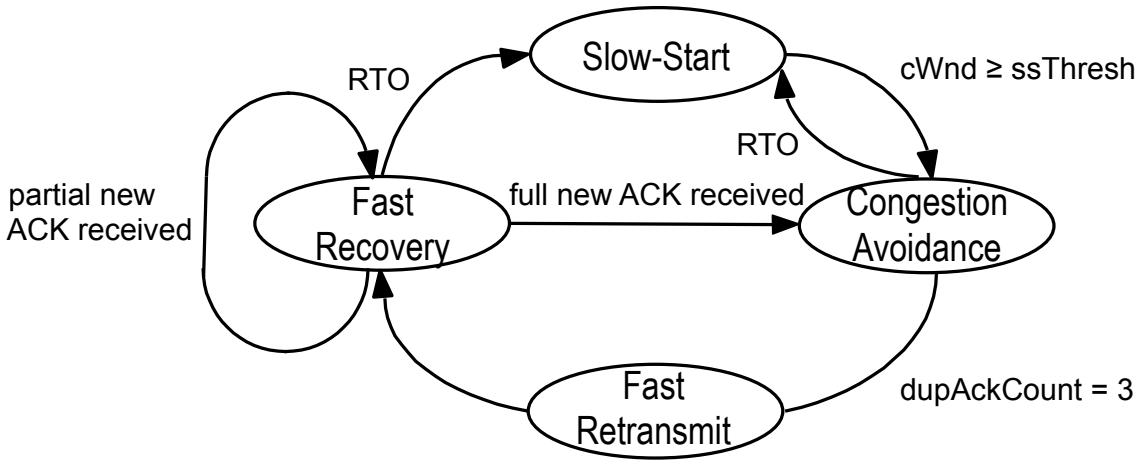

Figure 2.16. NewReno congestion control state transition diagram

2.3.4 TCP Vegas

As a delay-based congestion control algorithm, TCP Vegas [29] depends on packets' RTTs to detect congestion. While other loss-based algorithms such as Reno and NewReno are reactive, only invoking their congestion control scheme when a packet loss occurs, Vegas is more proactive by attempting to detect congestion before losses actually happen. Vegas continuously samples packet's RTT and compare it to the base RTT, which is the minimum RTT recorded during a connection lifetime. The base RTT along with the current congestion window value give the expected throughput while the sampled RTT together with the total number of bytes transferred during a sampling period allow Vegas to estimate the actual throughput. Vegas then uses the difference between the two throughput values to assess whether it is sending at the right rate. Since Vegas makes modifications on top of Reno, its congestion control state transition diagram is the same as Reno's. The Vegas algorithm is discussed again in terms of its implementation in Section 3.3.

2.3.5 TCP SACK

In this thesis, TCP SACK refers to the conservative SACK-based loss recovery algorithm described in [11]. Even though the SACK option (Section 2.2.2) has been widely deployed since its inception, the earlier implementations did not make a complete use of SACK information. Heavily based on Fall and Floyd's *pipe* algorithm for SACK [34], TCP SACK exploits the information carried by SACK blocks to assist the sender in making the right retransmission decisions and fully utilizing the available network's bandwidth. The result is an improvement in the overall TCP performance. In addition to NewReno's fast recovery, TCP SACK's

algorithm is another loss recovery algorithm developed to enhance TCP's performance in facing multiple losses. TCP SACK has the same congestion control state transition diagram as NewReno's. The algorithm is discussed in more depth in Section 3.2. This is also the algorithm we use to incorporate NAK and SNACK while performing comparison on different acknowledgment schemes.

2.3.6 TCP Westwood

TCP Westwood [22] is a sender-side modification to Reno's fast recovery algorithm to enhance TCP performance in heterogeneous networks. In a wireless environment in which corruption-based losses happen more often than congestion-based losses, the standard TCP variants (Tahoe, Reno, and NewReno) have no mechanisms to determine the real cause of a loss. They treat every loss as congestion-based and halve the congestion window causing a significant throughput degradation. Westwood, on the other hand, tries to estimate the current bandwidth based on the ACK rate and uses the estimation to adjust the sending rate when a loss happens. Unless a loss is due to network congestion, the bandwidth is not affected and the sending rate should not be reduced. The congestion control state transition diagram for Westwood is similar to Reno's.

2.3.7 TCP Westwood+

TCP Westwood+ [23] modifies Westwood's bandwidth estimation algorithm to reduce its aggressiveness in the presence of ACK compression [35]. Since Westwood samples the bandwidth on every ACK receipt, the spacing between ACK arrivals, which is alternated when ACKs are built up in the queue, impacts the correct estimation of the real bandwidth. To handle the problem, Westwood+

performs bandwidth estimation every RTT. Similar to Westwood, Westwood+ has the same congestion control state transition diagram as Reno's.

2.4 Other Related Work

Among all the acknowledgement schemes, SACK is the most popular in the research community with extensive studies. SACK is extended to Duplicate-SACK (D-SACK), which allows the receiver to report its receipt of duplicate segments caused by network's replication, packet reordering, ACK losses, or a premature RTO [36]. The D-SACK option has the same structure as SACK's, except its first block is dedicated to specify the sequence numbers of the reported duplicate packet. D-SACK is employed in Reordering-Robust TCP (RR-TCP) to improve TCP performance in networks that suffer significant packet reordering [37].

Improved SACK (ISACK) [38] is another modification that addresses the bit-efficient issue of SACK. Following a similar format as SACK, but instead of using two 32-byte sequence numbers to represent one isolated, contiguous chunk of data, ISACK uses a 1-byte offset and a 1-byte size fields to convey the same information. The development of ISACK also leads to the proposal of a new adaptive selective acknowledgement (ASACK) strategy that allows a dynamic switch between ISACK and SACK for performance enhancement purpose [38].

Another modification to SACK that consists of a small change in the SACK's behavior rather than the option format is called non-renegable SACK (NR-SACK). NR-SACK prevents the receiver from discarding SACKed data, reducing the amount of data the sender has to hold in its send buffer. When employing in Stream Control Transmission Protocol (SCTP) [39], the use of NR-SACK results in a better utilization of the send buffer's memory because the sender

can free up its buffer space before receiving cumulative ACKs [40]. When using with MPTCP [41], NR-SACK improves the overall throughput in the case where the send buffer's size becomes the bottleneck in the presence of multiple flows [42]. SACK motivated the development of another TCP variant called TCP k-SACK that exploits SACK's information in trying to detect congestion to improve throughput over wireless channels [43].

To enhance performance in the presence of multiple losses, other than the loss recovery mechanisms used in NewReno and the conservative TCP SACK, the Forward Acknowledgment (FACK) congestion control algorithm [44] performs congestion control while trying to recover from a loss by estimating the number of outstanding data using the receiver's forward-most data. The forward-most data represents the highest sequence number received correctly by the receiver. Another SACK-based approach proposed to replace the traditional fast recovery [19] is the rate-halving with bounding parameters algorithm [45].

Similar to SACK with different variants, the Vegas group consists of many versions: Vegas-A [46], AdaVegas [47], New Vegas [48], Vegas_M [49], Gallop-Vegas [50], Snug-Vegas [51], Vegas-W [52], Vegas-V [53], etc. The motivation behind these Vegas variants is to address the original Vegas problems, especially its fairness while competing for bandwidth resource with other flows in various network environments. Vegas fairness has been studied extensively [54–57].

To the best of our knowledge, there have been only a few works on comparison between the SACK, NAK, and SNACK as TCP acknowledgment mechanisms [14, 58, 59]. On the other hand, the amount of work related to comparison between different TCP congestion control algorithms both analytically and experimentally is numerous [60–64].

Chapter 3

Implementations

The standard ns-3 release comes with some existing TCP models, including Tahoe, Reno, and NewReno. As a part of our research work, we implemented Westwood and Westwood+ and contributed our code to the ns-3 community. We also performed some simulation analysis of the two protocols and compare their performance with others under different network scenarios to validate our implementations [65]. To accomplish the goals of this thesis, we performed additional implementations that are explained in this chapter. We begin this chapter by giving an overview of the TCP class structure in ns-3. We then present our implementation of the SACK-based loss recovery algorithm, followed by an explanation on how we incorporate NAK and SNACK into the algorithm. Finally, we discuss the implementation of TCP Vegas.

3.1 TCP Module and Class Interaction in ns-3

Residing in the Internet module that houses other protocols including IPv4, IPv6, and UDP, the implementation of TCP in ns-3 consists of multiple classes

communicating with each other to provide reliable transfer of data received from the underlying network to applications. We provide a brief discussion on each of the main classes from which new implementations are extended and illustrate their interaction in Figure 3.1 [66].

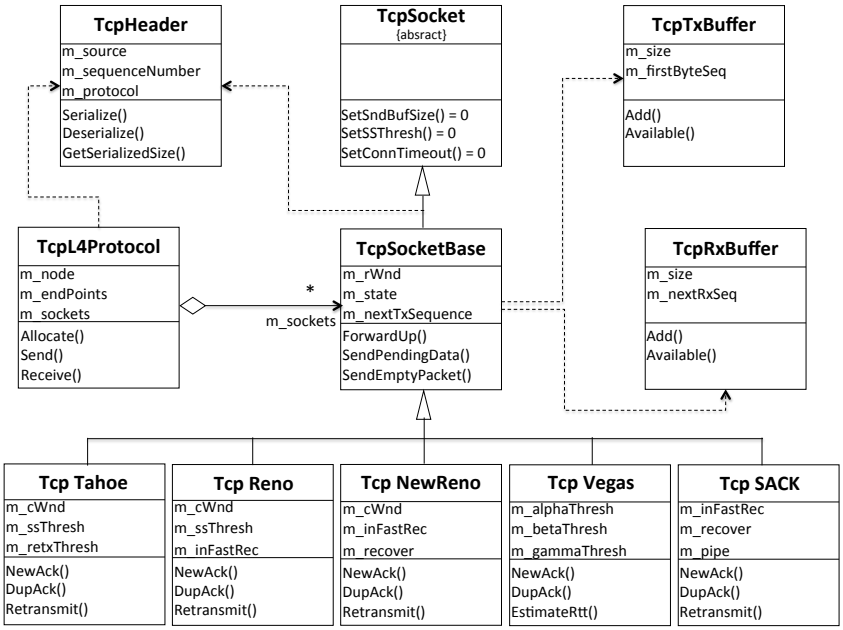

Figure 3.1. TCP class diagram in ns-3

- `TcpSocketBase`: This class contains the key TCP features including connection management and flow control, and a sockets interface to be called by the upper application layer. Inherited from `TcpSocket`, `TcpSocketBase` serves as the base for all TCP extensions.
- `TcpSocket`: This abstract class contains pure virtual functions for setting the essential TCP socket attributes that can be shared among different implementations.
- `TcpHeader`: This class implements the standard 20-byte TCP header.

- `TcpTxBuffer`: This class provides a buffer for the sender to store any data received from the application before sending it across the network. The buffer continues to hold the data segments until they are acknowledged.
- `TcpRxBuffer`: This class implements a buffer for the receiver to store and reorder data received from the network layer before passing it onto the application.
- `TcpL4Protocol`: Serving as an interface between the TCP socket and the network layer, the `TcpL4Protocol` class sends and receives packets to and from the network layer. It also performs checksum validation for incoming data.

3.2 Implementation of SACK, NAK, and SNACK

To obtain a fair comparison between the different acknowledgment mechanisms, we place NAK and SNACK in the loss recovery algorithm [11] that was designed to be used with SACK (refered to as TCP SACK in the thesis) to understand the impact of these options on the overall TCP performance. We choose this algorithm because it contains intelligent mechanisms that exploit the information carried by such TCP options. The different structures of NAK and SNACK in comparison to SACK require some modifications to the original algorithm that we will explain in the following subsections. While trying to incorporate NAK and SNACK into the algorithm, we make sure that the other parts of the algorithm are kept intact to preserve core congestion control principles [19, 67], and the spirits of NAK and SNACK as described in their original documents [3, 4, 15] are followed as closely as possible. We begin this section by explaining the TCP SACK implementation before transitioning into the discussion of NAK and SNACK in-

corporation.

3.2.1 SACK-Based Loss Recovery Algorithm

The section starts with a discussion on how the algorithm's important feature, the scoreboard, was implemented. It is then followed with the implementation of the main algorithm itself.

3.2.1.1 The Scoreboard

One of the important features of the algorithm, the scoreboard is a data structure used by the sender to keep track of SACK information. In this thesis, the scoreboard was implemented as a list of lists that contains the first sequence number of a sent segment m_sequenceNumber and a SACK flag m_isSacked as depicted in Figure 3.2. The sequence number is the key for iterating and searching through the scoreboard. The scoreboard consists of five main functions explained below. As a naming convention in ns-3, all global variables are named with a prefix m_ while those without the prefix are local variable or parameter names.

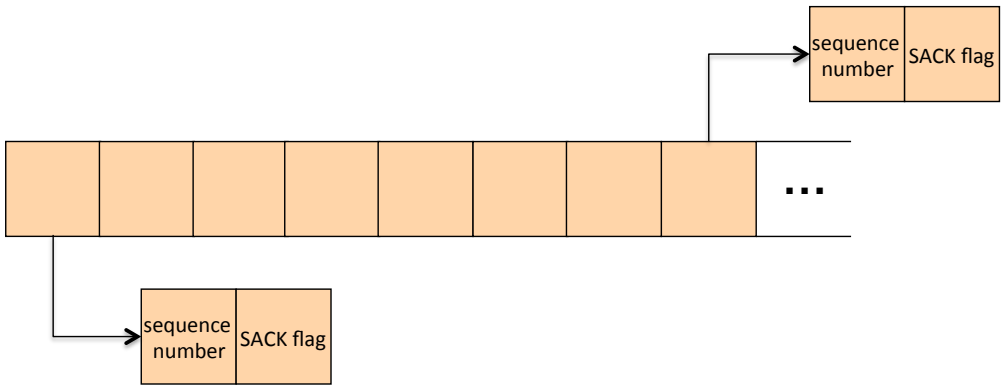

Figure 3.2. SACK scoreboard

- **Update**: Being called on the receipt of an ACK, **Update** takes the ACK number and the list of received SACK blocks as its arguments. **Update** discards all the ACKed segments (those segments whose initial sequence numbers are less than the ACK number) and sets the SACK flags for those reported in the SACK list as it iterates through the scoreboard. The function also updates m_highSack, a variable that keeps track of the highest sequence number having been SACKed.
- **IsLost**: This function determines whether a particular segment with the first sequence number seqNum has been lost. Based on the algorithm, the segment is considered lost if there exist dupThresh SACKed segments whose initial sequence numbers are greater than seqNum in the scoreboard.
- **SetPipe**: This function traverses the scoreboard to estimate the number of outstanding segments or those that are still in the pipe. An outstanding segment is the one whose initial sequence number falls between highAck and highData that satisfies one of the two conditions: (1) It has not either been SACKed or been determined to be lost; or (2) It has been retransmitted. While highAck keeps track of the highest sequence number that has been cumulatively ACKed, highData holds the highest sequence number that has been transmitted at a given time.
- **NextSeq**: This function determines the next should-be-transmitted sequence number. **NextSeq** implements a set of three rules that must be applied in the following order:

  Rule 1: This rule searches in the scoreboard for an unSACKed sequence number that is greater than highRxt and less than m_highSack and is determined to be lost by **IsLost**. highRxt holds the highest sequence number

that has been retransmitted. This rule allows the algorithm to recover from multiple losses due to a burst drop.

Rule 2: When Rule 1 fails, Rule 2 allows the algorithm to transmit up to one segment of newly unsent data stored in the sender's transmit buffer. With this rule, the algorithm can better utilize the available network's bandwidth by trying to keep the pipe full while recovering from losses.

Rule 3: In case both Rule 1 and Rule 2 fail, Rule 3 is applied to search for an unSACKed sequence number that is greater than highRxt and less than m_highSack. Unlike Rule 1, the sequence number returned by Rule 3 does need to be a lost sequence determined by **IsLost**. The goals are to maintain the ACK clock during the loss recovery phase and prevent the firing of the retransmission (RTO) timer that causes a severe performance drop due to the refilling of an empty pipe after another slow start phase.

3.2.1.2 The Algorithm

Using the information stored in the scoreboard, the sender responds to each ACK receipt based on the type of ACK through the two main functions **DupAck** and **NewAck**. Before going into the detail of these methods, we explain some global variables used in our implementation.

Global variables:

- m_cWnd represents the congestion window and is used by the sender to control its sending rate.
- m_ssThresh represents the slow-start threshold that marks the transition between the slow start and the congestion avoidance phases.

- m_retxThresh is the fast retransmit threshold that indicates the number of duplicate ACKs (dupACK) received before fast retransmit is triggered and is usually set to three [19]. This variable corresponds to the dupThresh parameter in the scoreboard's **IsLost** method.
- m_recover stores the highest transmitted sequence number before the sender enters fast recovery.
- m_highRxt stores the highest sequence number that has been retransmitted. This variable corresponds to highRxt when being passed to the scoreboard's member functions.
- m_highTxMark stores the highest sequence number that has been transmitted. This variable corresponds to highData when being passed to the scoreboard's member functions.
- m_pipe stores the number of segments that are still in transit and is the returned value of the scoreboard's **SetPipe**.

The DupAck method: As illustrated in Figure 3.3, upon the receipt of a dupACK, the sender calls its scoreboard's **Update** method to update the status of all stored segments. After the update, if the sender is currently in fast recovery phase, following Reno and NewReno's behaviors [19, 21], the SACK sender increases its m_cWnd by a segment size and sends more new data if the congestion window and the receiver's advertised window allow. When the sender is not trying to recover from any loss, the dupAck triggers a check on m_retxThresh. If the retransmit threshold has not reached, the **DupAck** method is exited with no additional actions performed. Otherwise, similar to Reno and NewReno, the sender enters fast retransmit, halving m_cWnd and retransmitting the missing segment.

The SACK sender then lets its conservative loss recovery algorithm governs the transmission of additional segments to make use of the available network's bandwidth in replacement of NewReno's fast recovery phase.

Before discussing SACK's method to recover from losses, it is important to mention its behavior when an RTO occurs. The expiration of an RTO is considered as an indication of a severe loss. Because comparing to NewReno, SACK transmits more data segments during its recovery phase. Hence, to prevent itself from being too aggressive, a SACK sender performs an additional action to make sure the loss that caused an RTO to fire is fully recovered before entering a new fast recovery. In the **DupAck** function, this is implemented through a quick comparison between the highest ACKed sequence number m_highAck and m_recover. In addition, if an RTO occurs while the sender is executing its *pipe* mechanism, the sender has to immediately terminate the algorithm.

Figure 3.4 illustrates SACK's loss recovery algorithm that we usually refers to the *pipe* technique throughout this thesis due to its origin [34]. This is the case in which the SACK sender tries to utilize the network's bandwidth by using the SACK information that it has been trying to keep track from the received SACK blocks. The loss recovery phase begins with a call to the scoreboard's **SetPipe** that estimates the number of segments currently in the network. The returned value is stored in m_pipe. The number of additional segments that the sender will transmit equals to the difference between m_cWnd (after being converted from bytes to segments) and m_pipe. The **NextSeq** function in the scoreboard determines which should be the next segment to be sent according to the three rules explained in the previous section.

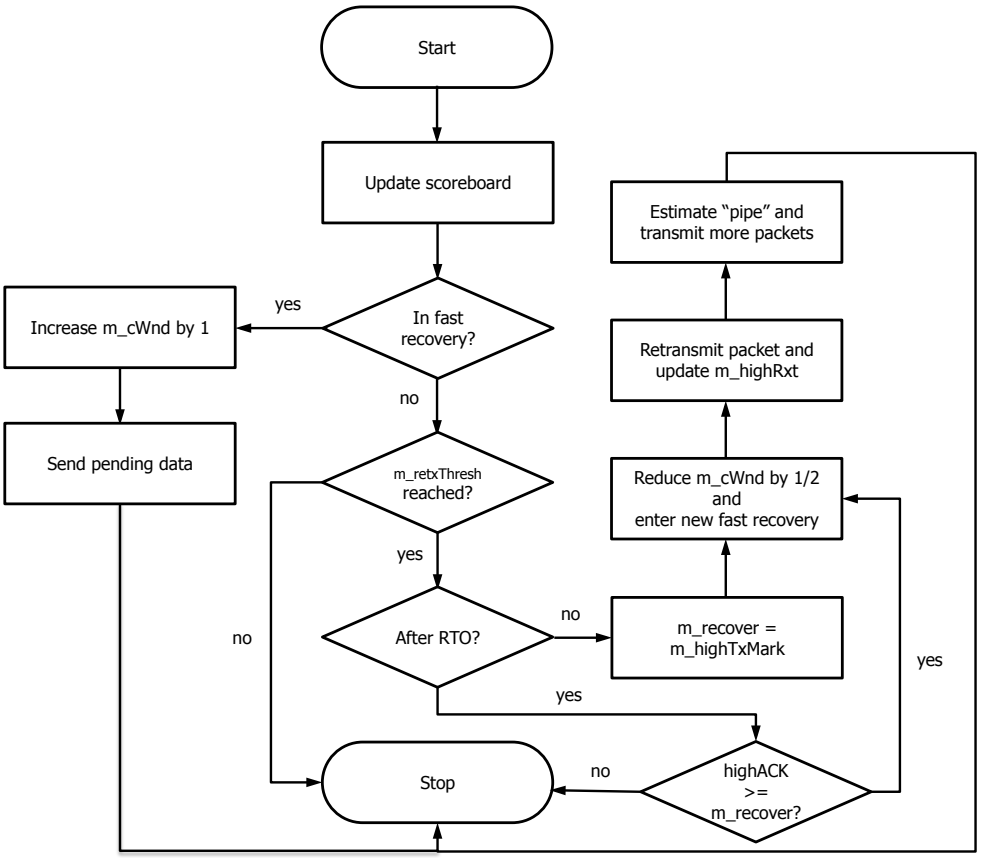

Figure 3.3. TCP SACK flowchart on receipt of a duplicate ACK

The NewAck method: As depicted in Figure 3.5, SACK's behavior on the receipt of a new ACK is very similar to NewReno's [21] except the addition of the pipe algorithm when a partial ACK is received.

3.2.2 NAK-Based Loss Recovery Algorithm

Based on the format of a NAK option and the TCP ACKing technique, we make two observations that allowed us to enhance NAK's capability beyond its original description [3].

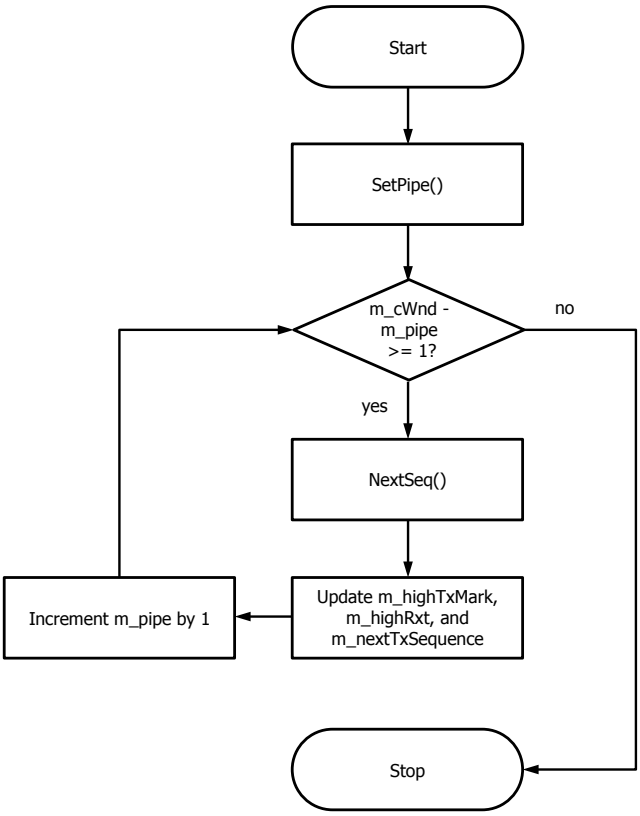

Figure 3.4. TCP SACK's pipe estimation flowchart

Observation 1: The arrival of a NAK option indicates that at least one segment immediately before the first segment and immediately after the last segment being NAKed in the option have left the network and arrived at the receiver successfully. That is, if the sender receives a NAK option that contains 4380 as the first byte being NAKed and 2 as the number of segments NAKed as in Figure 2.12, given the segment size of 1460 bytes, the sender can assume that the segments whose initial sequence numbers are 2920 and 7300 have been received.

If we assume that NAK options can be retransmitted, we have the second observation:

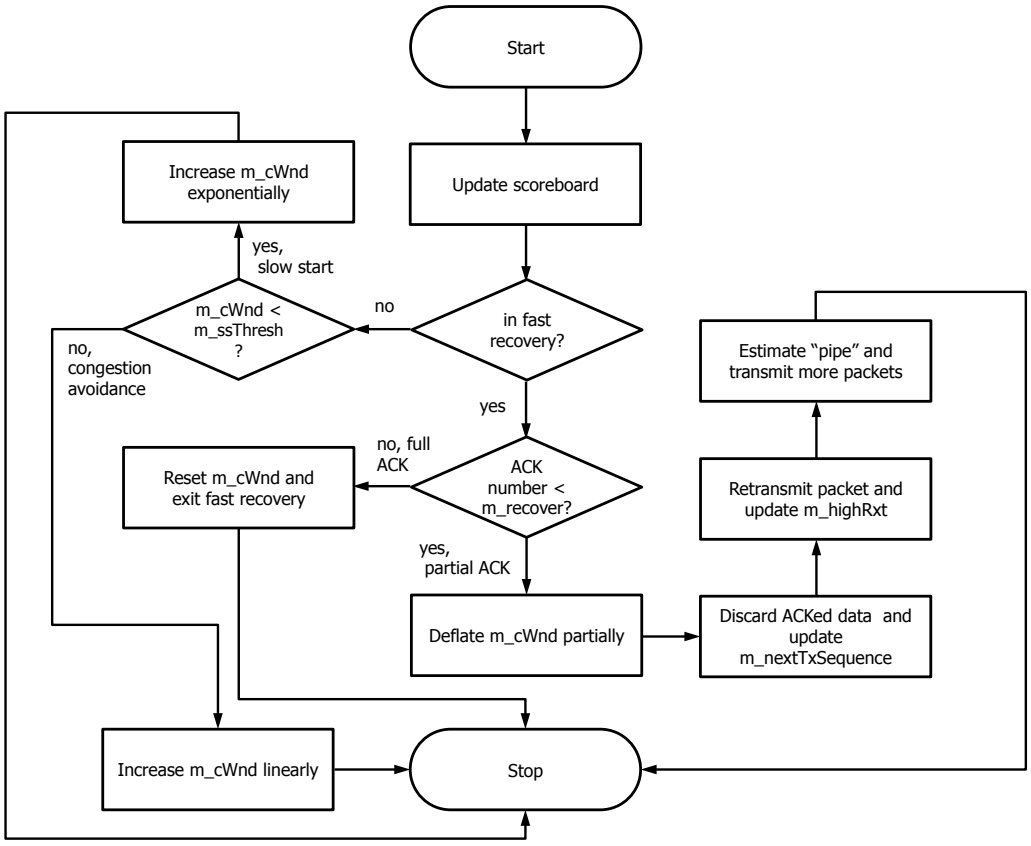

Figure 3.5. TCP SACK flowchart on receipt of a new ACK

Observation 2: Upon the receipt of two consecutive and distinct NAK options, the sender can assume that all segments between the two NAKs have left the network and arrived at the receiver. For example, if the sender receives the first NAK option that reports 1 missing segment with the first byte of 1460 and the second segment that reports another missing segment with the first byte of 7300, it can assume that all segments whose initial sequence numbers fall between the two NAKed bytes have been received.

Following the SACK design philosophy, we say that those segments in our observations as being implicitly SACKed. The observations also suggest an implementation of the NAK-based loss recovery algorithm with two different modes:

unreliable and quasi-reliable. In the unreliable mode, NAK options are not retransmitted. In this case, only the first observation holds. In the quasi-reliable mode, retransmission of NAK options up to a certain threshold is allowed, increasing the chance for at least one copy of an option is received by the sender. With this assumption, both observations hold.

3.2.2.1 The Scoreboard

As shown in Figure 3.6, the NAK scoreboard contains two flags in addition to the sequence number field, a NAK m_isNaked and an ISACK m_isISacked flag. A set NAK flag indicates that the corresponding segment has been reported as missing while a segment with an ISACK flag set to true is assumed to have left the network based on the previous observations. ISACKed data is similar to SACKed; they remain in the scoreboard until they are explicitly ACKed by the acknowledgment number, but they should not be retransmitted.

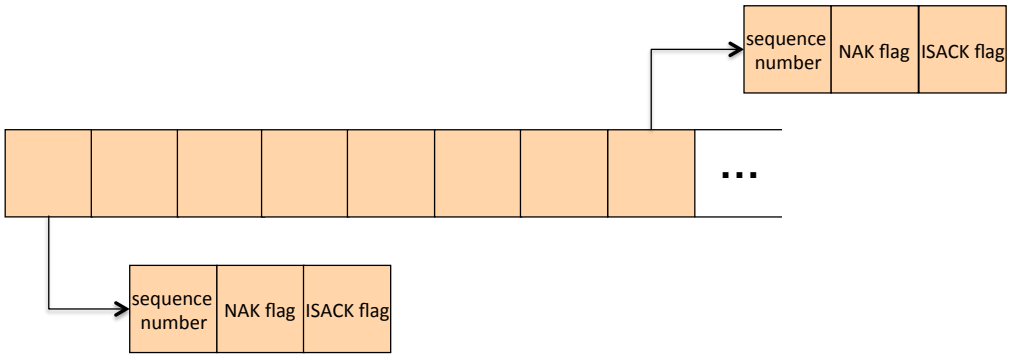

Figure 3.6. NAK scoreboard

The implementation of NAK scoreboard includes three main functions:

- **Update**: Similar to the SACK scoreboard, this function discards all ACKed segments and updates all the NAK and ISACK flags depending on the NAK

transmission mode. m_highNak that holds the highest NAKed sequence number is also kept track.

- **SetPipe**: To determine the amount of outstanding data, this function scans the scoreboard to search for any segment whose initial sequence number falls between highAck and highData that satisfies one of the two conditions: (1) it has not been NAKed or implicitly SACKed, or (2) it has been retransmitted.

- **NextSeq**: Following the SACK algorithm, this function implements the three rules for determining the next transmitted sequence, in which the first and the third rules are slightly different from those in SACK while the second rule is the same:

  Rule 1: This rule searches for a segment in the scoreboard whose initial sequence number falls between highRxt and m_highNak that has been NAKed (thus the ISACK flag is not set).

  Rule 3: Similar to rule 1, except that the NAK flag needs not be set.

3.2.2.2 The Algorithm

Given the two NAK modes, we implemented an additional 3-byte option NakMode that is appended to the SYN-ACK message during the connection establishment. The option allows the receiver to notify the sender which NAK mode it is going to use during the connection so that the sender knows how to update its scoreboard accordingly. The option contains a 1-byte option type field, a 1-byte length field, and a 1-byte mode field. For our experiment, we use 253 as the NakMode type. The rest of the algorithm is similar to the SACK algorithm discussed in the previous section.

3.2.3 SNACK-Based Loss Recovery Algorithm

The implementation of the SNACK-based loss recovery algorithm is a combination of the previous two algorithms. The scoreboard for SNACK is depicted in Figure 3.7.

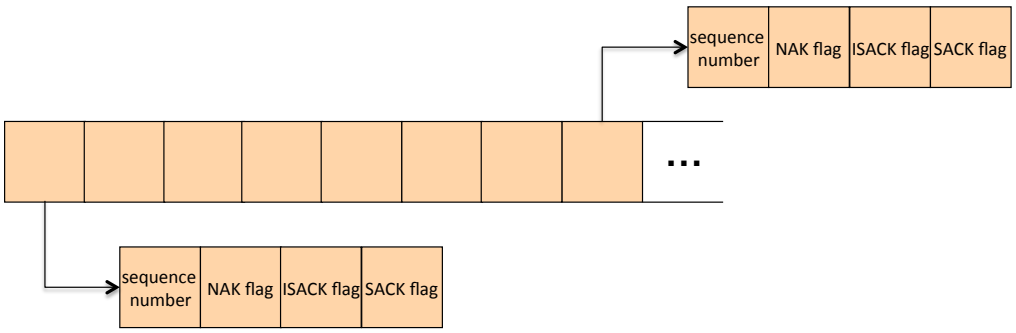

Figure 3.7. SNACK scoreboard

3.3 Implementation of TCP Vegas

The implementation of TCP Vegas in this thesis follows the Linux kernel's implementation [68] with the following key differences when comparing to the original Vegas paper [29] and the original implementation under the University of Arizona x-kernel framework [69]:

- The retransmission mechanism described in the paper is not implemented. Similar to Linux, ns-3 already uses fine-grained timers that are deployed in all ns-3's TCP variants including Tahoe, Reno, and NewReno.

- Instead of increasing the congestion window every other RTT during slow start phase as suggested by the authors in the paper, this implementation

adjusts the window every RTT as in other variants to avoid performance penalty.

- To calculate the actual throughput, minimum RTT sample is used to prevent the impact of delayed ACKs on the measurements.

3.3.1 Global Variables

Our Vegas implementation consists of the following important global variables:

- m_alpha is the threshold on the lower bound of packets in the network used to adjust m_cWnd during congestion avoidance phase.
- m_beta is the threshold on the upper bound of packets in the network used to adjust m_cWnd during congestion avoidance phase.
- m_gamma is the threshold used to limit the exponential increase of m_cWnd during slow start phase.
- m_baseRtt keeps track of the minimum of all RTT measurements during the whole connection. This value is the propagation delay.
- m_minRtt keeps track of the minimum RTT measured during a Vegas cycle to find the current propagation delay and queuing delay.
- m_cntRtt is the number of RTT measurements taken during a Vegas cycle.
- m_begSndNxt stores the right edge of the window at the beginning of a Vegas cycle.
- m_cntBytes stores the number of data bytes transmitted during a Vegas cycle.

- m_doingVegasNow is a boolean variable that is set to TRUE during a Vegas cycle, starting from the time a distinguished segment is sent to the time that segment is acknowledged.

3.3.2 Algorithm

A Vegas cycle begins with the transmission of a distinguished segment and ends when that segment is acknowledged by an ACK. Every time a data packet is sent, the sender checks the value of m_doingVegasNow. If m_doingVegasNow is TRUE, a cycle is already in action and all we need to do is to update m_cntBytes by adding the size of the transmitted packet to it. Otherwise, when m_doingVegasNow is FALSE, a new Vegas cycle will begin with the recording of the window's right edge into m_begSndNxt. The sender uses this value to detect the arrival of the ACK for the distinguished segment. Furthermore, the sender sets m_doingVegasNow to TRUE and starts counting m_cntBytes. Since Vegas is a delay-based congestion control algorithm, accurate RTT measurement plays an important role. Upon the receipt of an ACK, the sender also needs to update m_baseRtt and m_minRtt. While m_minRtt is reset every Vegas cycle, m_baseRtt is never reset during a connection lifetime.

Figure 3.8 shows the steps that TCP Vegas takes upon the receipt of a new ACK. For every new ACK that arrives, Vegas checks to see if it can terminate its current cycle by comparing the ACK sequence number and m_begSndNxt. If the ACK sequence number is greater than m_begSndNxt, the distinguished segment is acknowledged and the cycle is completed. Vegas then checks to see if it has taken enough RTT samples during the operation of the cycle. The reason Vegas sets a threshold (which is 3 in our implementation) on the RTT samples is to avoid the

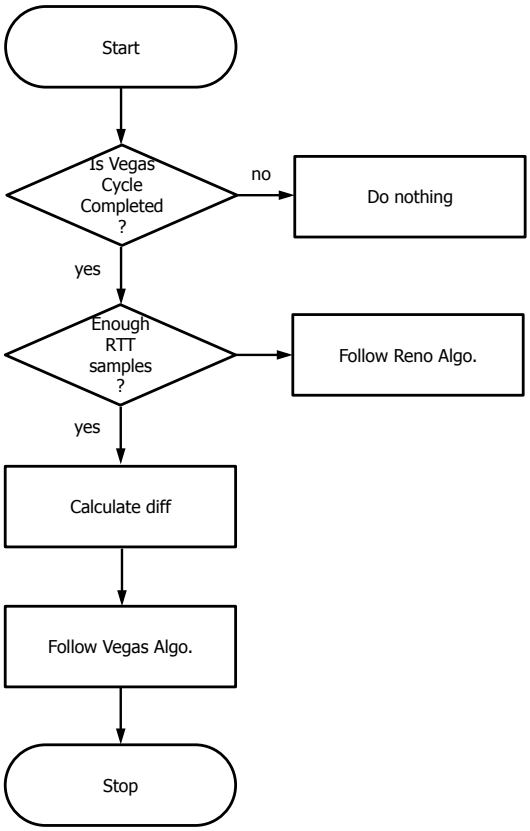

Figure 3.8. TCP Vegas flowchart on receipt of a new ACK

impact of delayed ACK on the RTT values. If this condition is met, Vegas triggers its own congestion algorithm. Otherwise, it follows the normal Reno congestion algorithm with exponential or linear window increase depending on the current phase. Once Vegas uses its algorithm, it calculates the difference diff between the expected sending rate and the actual rate as shown in Equation 3.1 and uses diff along with the 3 threshold values, m_alpha, m_beta, and m_gamma, to estimate and control the amount of extra data being sent to the network. Specifically, Vegas switches from slow start mode to congestion avoidance mode whenever the actual rate falls below the expected rate by m_gamma, that is when diff exceeds m_gamma. While in congestion avoidance mode, Vegas compares diff with m_alpha

and m_beta. m_alpha corresponds to having too little extra data packets in the network while m_beta corresponds to having too much data. Hence, when diff is smaller than m_alpha, Vegas increments its m_cWnd by 1 segment size to speed up its sending rate. When diff is greater than m_beta, Vegas decrements its m_cWnd by 1 segment size to avoid overflowing the network.

$$\mathsf{diff} = \mathsf{expected} - \mathsf{actual} \tag{3.1}$$

where

$$\mathsf{expected} = \mathsf{m\_cWnd}/\mathsf{m\_baseRtt} \tag{3.2}$$

$$\mathsf{actual} = \mathsf{m\_cntBytes}/\mathsf{m\_minRtt} \tag{3.3}$$

Chapter 4

Results and Analysis

In this section, we present the results obtained from simulating the different acknowledgment schemes and congestion control algorithms under various scenarios and then analyze the behaviors of each mechanism. The error control analysis covers the normal TCP ACKing method as implemented in NewReno, the selective ACK in TCP SACK, the negative ACK in our TCP NAK, and the selective-negative ACK in our TCP SNACK. For congestion control analysis, we study 7 different algorithms including Tahoe, Reno, NewReno, Vegas, Westwood, Westwood+, and TCP SACK. We also explain in details the topology and parameters used in each simulation scenario.

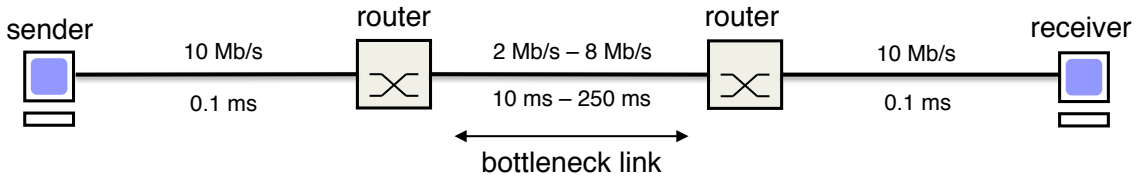

Figure 4.1. Single flow topology

4.1 Error Control Results and Analysis

Parameter	Values
Access link bandwidth	10 Mb/s
Bottleneck link bandwidth	2 Mb/s to 8 Mb/s
Access link propagation delay	0.1 ms
Bottleneck link propagation delay	10 ms to 250 ms
Packet MTU size	1500 B
Delayed ACK count	2 segments
Delayed ACK timeout	200 ms
Error model	BurstErrorModel
Burst error rate	10^{-7} to 10^{-2}
Burst size	1 to 4
Application type	Bulk send application
Simulation time	5000 s
Number of runs	10

Table 4.1. Simulation parameters for ACK mechanisms tests

To study the four acknowledgment mechanisms, we use a topology that consists of a single source and a single sink interconnected through two gateways as depicted in Figure 4.1. We refer the two endpoint-router links as the access links and the router-router link as the error-prone, long-delay bottleneck link. All links are constructed using `ns3::PointToPointHelper` class while errors are introduced into the bottleneck link using the `BurstErrorModel`. The `BurstErrorModel` that we implement as part of this thesis determines a burst of packets as being errored based on the burst rate and the burst size. The burst rate specifies the spacing between error events while the burst size determines the number of packets being flagged as errored at each error event. The burst error rate ranges from 10^{-7} to 10^{-2}. Each access link has a bandwidth of 10 Mb/s and a negligible propagation delay of 0.1 ms. The bottleneck link takes the bandwidth values of 2 Mb/s to 8 Mb/s and its delay is varied from 10 ms to 250 ms (equivalent to 20 ms to 500 ms

RTT). We use `BulkSendApplication` to generate traffic across the network with an MTU size of 1500 bytes. Each simulation has a 5000 second duration and is run 10 times to achieve a 95% confidence interval. These simulation parameters are summarized in Table 4.1.

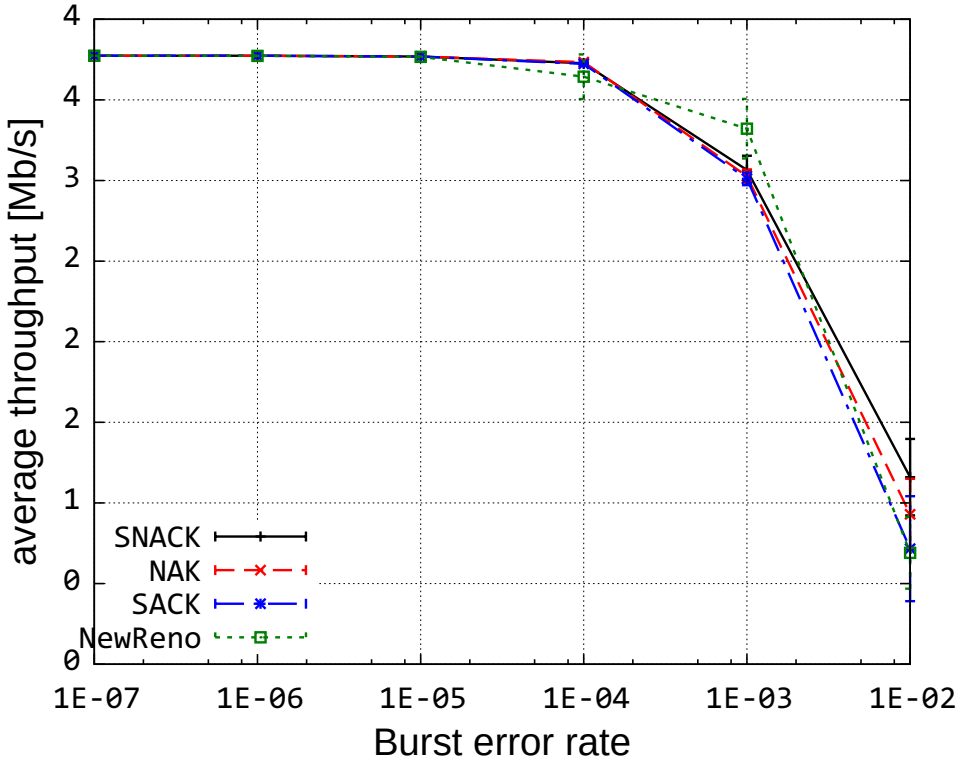

Figure 4.2. Throughput vs. increasing burst error rate

Figure 4.2 shows the throughput of the protocols as the burst rate increases. In this scenario, the bottleneck link has a bandwidth of 4 Mb/s and a delay of 50 ms. Overall, with the increasing error rate, the performance of all protocols degrade due to the high number of retransmissions required. When the error reaches 10^{-2}, SNACK performs the best. The bit vector allows SNACK receiver to convey more information about its buffer state per SNACK packet than the other protocols, which helps fasten the recovery process.

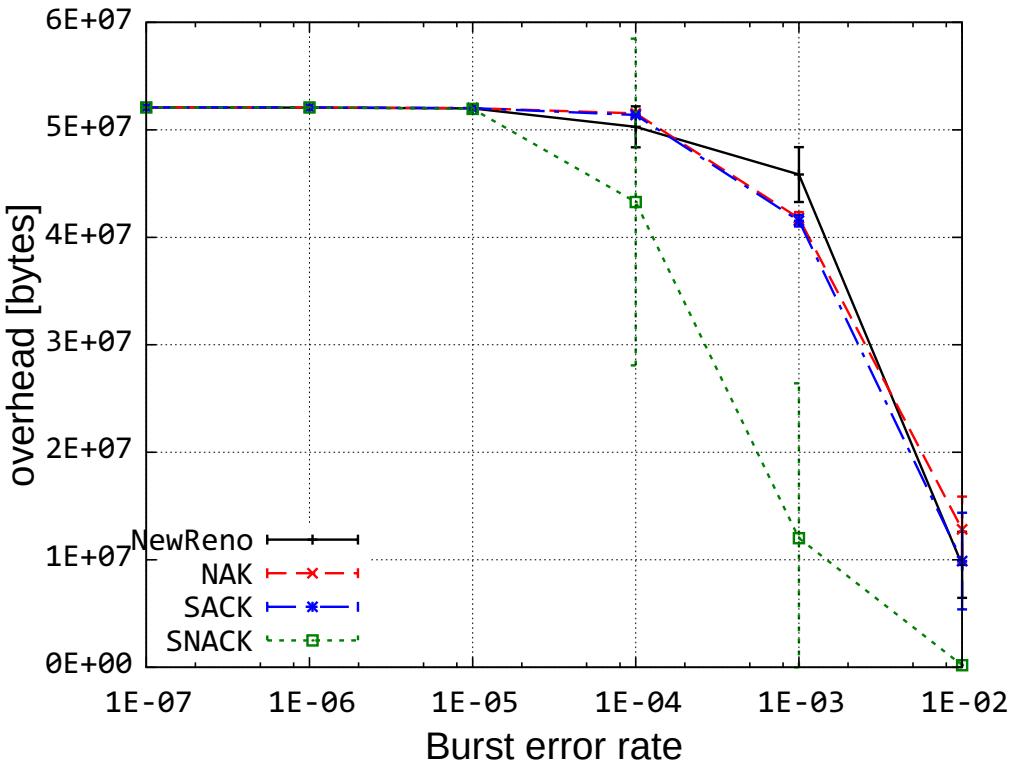

Figure 4.3. Overhead vs. increasing burst error rate

Figure 4.3 shows the overhead of the protocols with the increasing error rate. With its compact option format and its ability to carry more information in the option, SNACK has the lowest overhead. While it costs 24 bytes for a SACK packet to inform 3 isolated data chunks in the receiver buffer, a 1-byte bit vector in SNACK can inform more than 3 such chunks. Although a NAK option is only 7 bytes in length, the small NAK packet size when compared to SACK does not lower NAK overhead because a NAK option can only inform a single hole in the receiver buffer. When the error rate is high, more packets are corrupted, a NAK receiver needs to generate more NAK packets resulting in a high cumulative overhead.

4.2 Congestion Control Results and Analysis

In this part, we simulate Tahoe, Reno, NewReno, Vegas, SACK, Westwood, and Westwood+ under 2 different network environments. In the first scenario, we set up a single flow topology similar to the one depicted in Figure 4.1 in which the bottleneck link is prone to errors. We study how well the protocols handle corruption-based losses that are very common in wireless networks using throughput as the performance metric. The access links have a bandwidth of 100 Mb/s while the bottleneck link bandwidth ranges from 10 Mb/s to 50 Mb/s on an increment of 10 Mb/s. The delay on the access links are negligible and the delay on the bottleneck link is varied from 50 ms to 250 ms (100 ms to 500 ms RTT). The packet error rates (PER) are generated using `RateErrorModel` and have a range from from 10^{-7} to 10^{-2}. Due to the randomness in the error generation, each simulation which has a duration of 600 seconds is replicated 20 times to achieve a 95% confidence interval. These simulation parameters are summarized in Table 4.2.

Parameter	**Values**
Access link bandwidth	100 Mb/s
Bottleneck link bandwidth	10 Mb/s to 50 Mb/s
Access link propagation delay	0.1 ms
Bottleneck link propagation delay	10 ms to 250 ms
Packet MTU size	1500 B
Delayed ACK count	2 segments
Delayed ACK timeout	200 ms
Error model	RateErrorModel
Packet error rate	10^{-7} to 10^{-2}
Buffer size	BDP
Application type	Bulk send application
Simulation time	600 s

Table 4.2. Simulation parameters for single flow test on TCP protocols

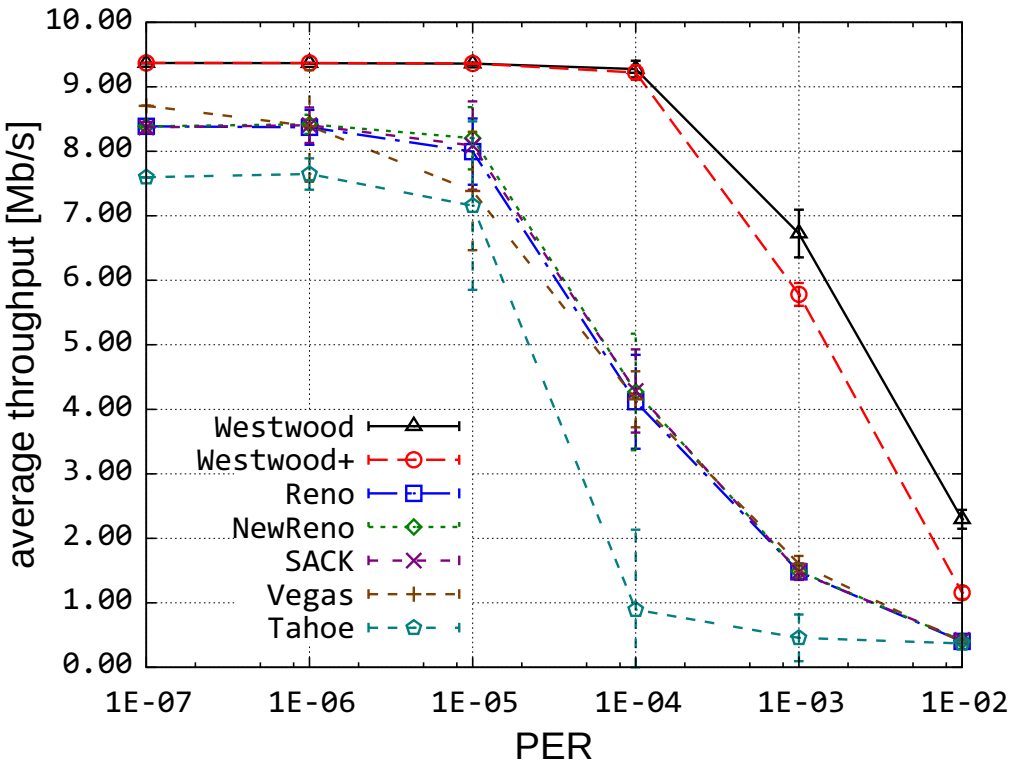

Figure 4.4. Average throughput vs. packet error rate

Figure 4.4 plots the average throughput as the packet error rate increases from 10^{-7} to 10^{-2}. In this scenario, the bottleneck link has a bandwidth of 10 Mb/s and a delay of 100 ms. When the error rate is high causing more packet drops, all protocols have to retransmit missing data to provide reliable service, resulting in a throughput degradation. Reno, NewReno, and SACK behave similarly. They treat all packet losses as congestion-based and halve their sending rate upon the receipt of 3 duplicate ACKs. Hence, comparing to Westwood and Westwood+, Reno, NewReno, and SACK achieve much lower throughput even when the error rate is as low as 10^{-7}. Their throughput is halved (from 8 Mb/s to 4 Mb/s) when the PER increases from 10^{-5} to 10^{-4} and continues to drop to about 0.5 Mb/s when the PER reaches 10^{-2} due to them continuing to reduce their sending

rate unnecessarily. Furthermore, with the increasing error rate, the number of retransmission timeouts also increases because retransmitted packets have a high possibility of being corrupted. This causes the sender to transfer back to slow-start phase during which it has to refill the transmission pipe. Tahoe performs worse than Reno and NewReno because the Tahoe sender switches back to slow start even when it receives 3 duplicate ACKs. On the other hand, both Westwood and Westwood+ outperform all the other protocols. While Tahoe, Reno, NewReno, and SACK blindly halve their sending rates upon a loss or switch back to slow start when an RTO expires, Westwood and Westwood+ try to estimate the network's bandwidth and use the estimated value to adjust its sending rate. The network's bandwidth is not affected unless congestion occurs. Hence, Westwood and Westwood+ are able to maintain their high throughput when all losses are due to packet corruptions. Westwood and Westwood+ achieve a high throughput of about 9.3 Mb/s when the PER is 10^{-7} and their throughputs are not degraded until the PER reaches 10^{-4}. When the PER is above 10^{-4}, Westwood performs better than Westwood+. While Westwood samples the bandwidth every received ACK, Westwood+ performs its sampling every RTT. The higher sampling interval takes Westwood+ a longer time to stabilize to the correct bandwidth when comparing to Westwood. In the presence of high error rate, there is a higher chance for 3 duplicate ACKs or an RTO to occur before Westwood+ stabilizes, causing the use of a low, incorrect bandwidth estimate in adjusting the sending rate.

Figure 4.5 plots the average throughput as the bottleneck speed increases from 10 Mb/s to 50 Mb/s while the PER is fixed at 10^{-5} and the bottleneck delay is 100 ms. Both Westwood and Westwood+ are able to utilize the available bandwidth better than the other protocols. For the standard TCP variants such as

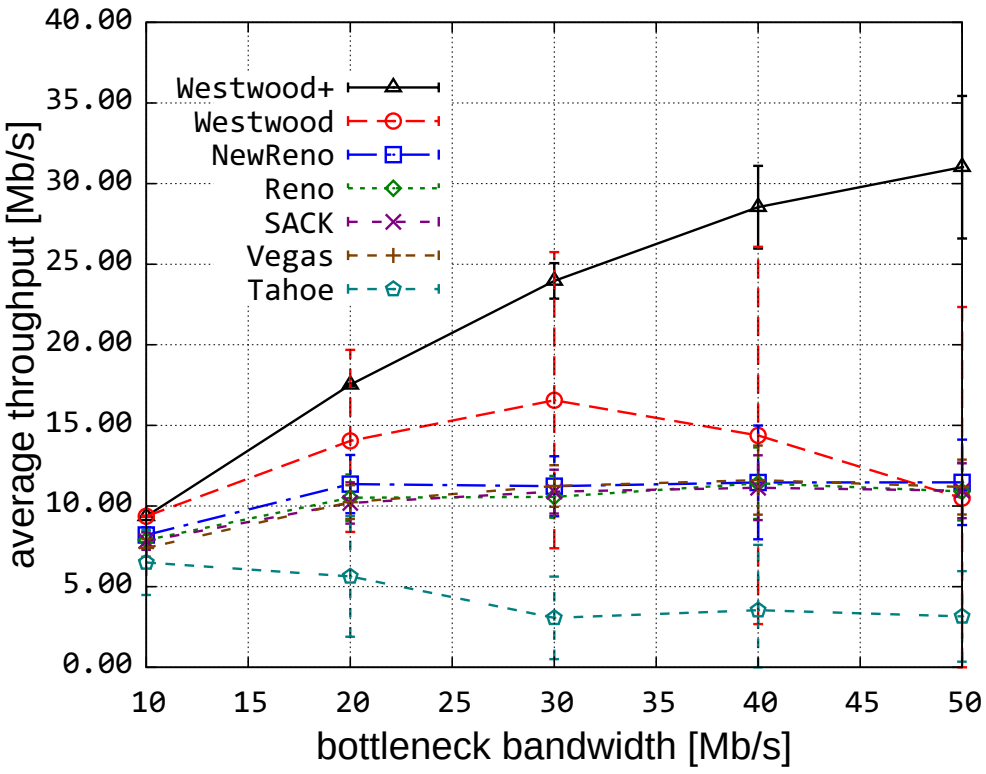

Figure 4.5. Average throughput vs. bottleneck speed

Tahoe, Reno, and NewReno, in a steady-state environment, the average congestion window w is inversely proportional to the square root of the packet loss rate p as given in Equation 4.1 [25].

$$w = \frac{1.2}{\sqrt{p}} \tag{4.1}$$

From Equation 4.1, the packet loss rate p is roughly $\frac{1.5}{w^2}$. Furthermore, Equation 4.2 specifies the average congestion window w required for a TCP flow with round-trip time R in seconds and packet length of D bytes to achieve an average throughput of B b/s [25].

$$w = \frac{BR}{8D} \tag{4.2}$$

A packet loss rate p implies that there is at most 1 loss every $\frac{1}{p}$ packets, which is equivalent to at most 1 loss event every $\frac{1}{pw}$ RTTs. Combining this result with Equations 4.1 and 4.2, we have Equation 4.3 that specifies the number of RTTs between loss events required to achieve an average throughput of B in b/s [25].

$$\text{RTTs between losses} = \frac{BR}{12D} \tag{4.3}$$

Using Equations 4.1, 4.2, and 4.3, we derive Table 4.3 that calculates the number of RTTs between losses, the average congestion window w, and the packet loss rate p corresponding to each value of the bandwidth B from 10 Mb/s to 50 Mb/s. Here, our R is 0.2 seconds and our D is 1500 bytes.

Throughput (Mb/s)	RTTs between losses	w (segments)	p
10	111.1	166.7	5×10^{-5}
20	222.2	333.3	10^{-5}
30	333.3	500	6×10^{-6}
40	444.4	666.7	3×10^{-6}
50	555.6	833.3	$2\times10^{-}6$

Table 4.3. Performance of standard TCP in steady state

From Table 4.3, with our PER being fixed at 10^{-5}, Tahoe, Reno, and NewReno are unable to fully utilize the link, especially when the bottleneck bandwidth is equal to or greater than 30 Mb/s. At 10 Mb/s, Reno and NewReno can achieve a throughput of about 8 Mb/s whereas Tahoe throughput is much smaller because its lack of the Fast Recovery phase that helps prevent the transmission pipe from being drained completely after a loss. When the available bandwidth increases from 10 Mb/s to 20 Mb/s, Reno and NewReno throughputs are improved, even

though not as much as we expect based on our theoretical calculation in Table 4.3. However, after that, Reno and NewReno throughputs stay almost constant when the available bandwidth goes above 20 Mb/s. In order for Tahoe, Reno, and NewReno to achieve higher throughput in our testing scenario, the PER has to be smaller than 10^{-5}, or at most 10^{-6} as indicated in the table. On the other hand, because Westwood and Westwood+ actually estimate the network's bandwidth and use it to adjust the sending rate, they perform better than the other protocols. While Westwood+ continues to improve its throughput as more bandwidth is available, Westwood behavior is quite unstable, especially when the bottleneck speed is higher than 10 Mb/s. Performing the sampling based on ACK inter-arrival times causes Westwood to overestimate the available bandwidth, resulting in lots of packet drops at the queue. In addition to retransmitting packets that are dropped due to errors, Westwood has to retransmit many packets that are dropped due to queue overloading. Based on our statistics, the average number of Westwood's retransmitted packets is about three times larger than those in Westwood+ when the bottleneck bandwidth exceeds 10 Mb/s.

Figure 4.6 plots the average throughput achieved by each of the protocols when the bottleneck delay increases from 50 ms to 250 ms. In this scenario, the PER is fixed at 10^{-5} and the bottleneck speed is 50 Mb/s. Overall, when the network delay is high, it takes longer for a packet to arrive at the destination. It also takes longer for the sender to receive feedback from the other end, which in turn delay the updating of the sending rate. We again see the superior performance of Westwood+ over the other protocols. We also notice that with 50 Mb/s speed, Westwood performs even worse than Reno and NewReno when the delay exceeds 100 ms.

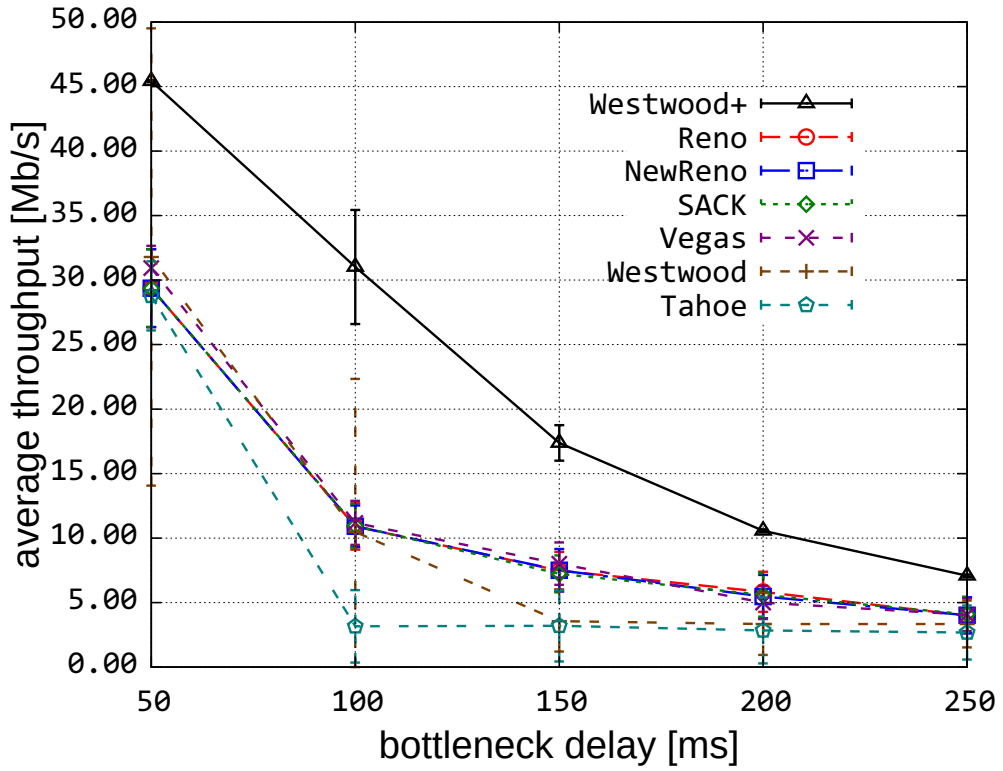

Figure 4.6. Average throughput vs. bottleneck delay

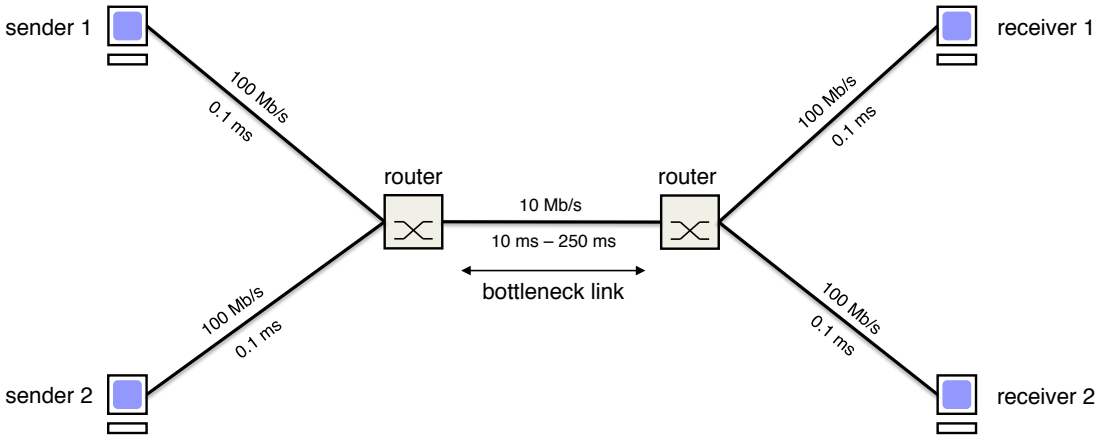

Figure 4.7. Dumbbell topology

In the second set of experiments, we study the performance of the protocols in a congested environment. We set up a dumbbell network topology as illustrated in Figure 4.7. At each edge of the network are two nodes serving as the sources at one end and the sinks at the other end. Traffic across the network is generated using `BulkSendApplication` with an MTU size of 1500 bytes. At the core of the network are two nodes serving as the routers that are interconnected through a bottleneck link with a fixed bandwidth of 10 Mb/s and a delay varied from 10 ms to 250 ms (20 ms to 500 ms RTT). The buffer size of the bottleneck link is set to the bandwidth $\times$ delay product (BDP). All the access links that connect the endpoints with the routers have a bandwidth of 100 Mb/s. Each simulation generates two flows of traffic that may or may not use the same TCP variant, depending on our testing scenarios. Since NewReno is the current standard TCP, it is used as the baseline in any scenarios that require different TCP variants on different flows. The duration of each simulation is 600 seconds. To reduce the impact of synchronization and phase effects, especially when Drop Tail queues are used at the bottleneck, we let the 2 flows start at different times and vary the start time of the second flow and average the results. Simulation parameters are summarized in Table 4.4. We evaluate the congestion control protocols based on the following properties: intra-protocol fairness, RTT fairness, friendliness, and link utilization.

4.2.1 Jain's Fairness index

We evaluate the fairness and friendliness of the protocols using the Jain's fairness index metric [70]. The index is computed using Formula 4.4 where x_i denotes the throughput of flow i and n denotes the total number of flows. The

Parameter	Values
Access link bandwidth	100 Mb/s
Bottleneck link bandwidth	10 Mb/s
Bottleneck link propagation delay	10 ms to 250 ms
Packet MTU size	1500 B
Delayed ACK count	2 segments
Application type	Bulk send application
Queue type	Drop tail
Queue size	BDP
Simulation time	600 s

Table 4.4. Parameters for congestion control simulations

index can take any value in the [0,1] interval with 1 showing the best fairness.

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2} \tag{4.4}$$

4.2.2 Link Utilization

We measure how well the protocol can utilize the available bottleneck bandwidth by simulating two flows of the same TCP variants. The plot in Figure 4.8 illustrates the ratio between the total throughput and the bottleneck bandwidth as the bottleneck delay increases from 10 ms to 250 ms.

Overall, for all protocols, the high delay causes the throughput of both flows to be dropped, which is reflected through the degradation in the total utilization. At a small delay (less than 100 ms), all protocols can achieve more than 90% utilization of the link capacity. Westwood suffers the most as delay increases. Because Westwood determines its sending rate based on ACK receipts, the longer the delay, the longer the ACK inter-arrival times, which results in the lower estimated bandwidth.

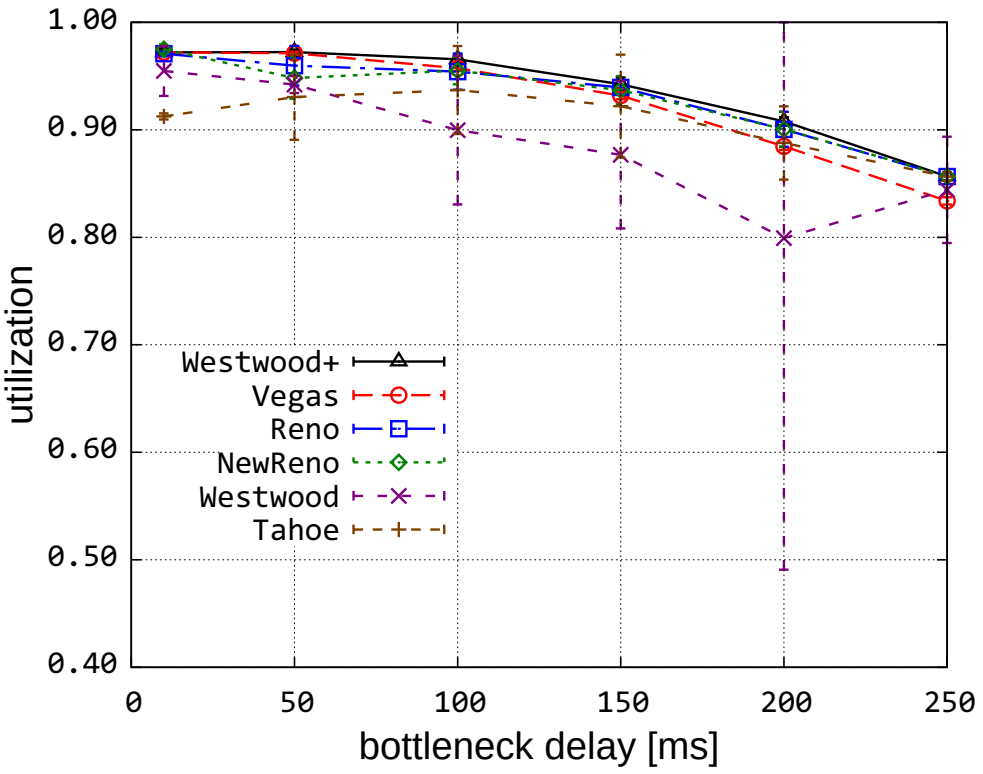

Figure 4.8. Utilization vs. increasing bottleneck delay

4.2.3 RTT Fairness

We measure the fairness in sharing the bottleneck bandwidth of two flows running the same variant but having different delays. While the first flow's delay varies from 10 ms to 250 ms, the second flow's is fixed at 250 ms. In this scenario, the bottleneck delay is set to 100 ms. We compute Jain's fairness index using the throughput obtained from each flow.

Figure 4.9 shows that the closer the second flow's delay to the first flow's, the better the fairness, especially when second flow's delay reaches 250 ms, all protocols achieve the greatest fairness of 1. Vegas is fairer than all the other protocols, which is explained by the unaggressiveness of Vegas algorithm. While

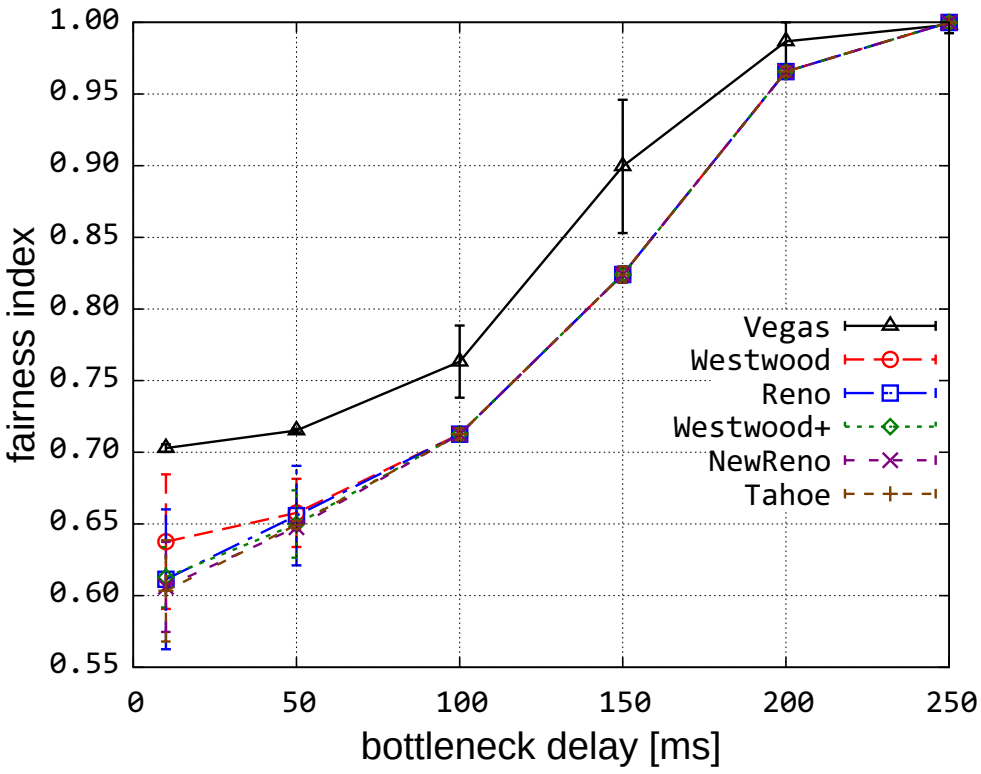

Figure 4.9. RTT fairness vs. increasing second flow's delay

the other protocols always try to throttle the link to get feedback on network condition, Vegas tries to anticipate congestion using network's delay. Hence, a Vegas flow rarely causes a loss by trying to overwhelm the queue.

4.2.4 Friendliness

We measure how friendly the protocols are when trying to compete with TCP NewReno over a bottleneck link with varying delays. The plot in Frigure 4.10 shows that Tahoe, Reno, and Westwood+ are very friendly when competing with NewReno. Actually, unless we simulate Reno and NewReno under a scenario where multiple losses per sending window occur, they should behave very similar to each other. Westwood and Vegas have the most interesting behavior in this

case. Vegas has the lowest fairness index throughout the whole simulation because Vegas bandwidth is stolen by the much more aggressive NewReno flow. In our simulations, we set the queue size to be the bandwidth × delay product, hence it is proportional to the bottleneck delay. At small delay values, the queue sizes are also small. However, due to its way of adjusting the sending rate upon a loss, Westwood causes more losses, especially with small queue sizes, resulting in its unfriendly behavior when competing with NewReno.

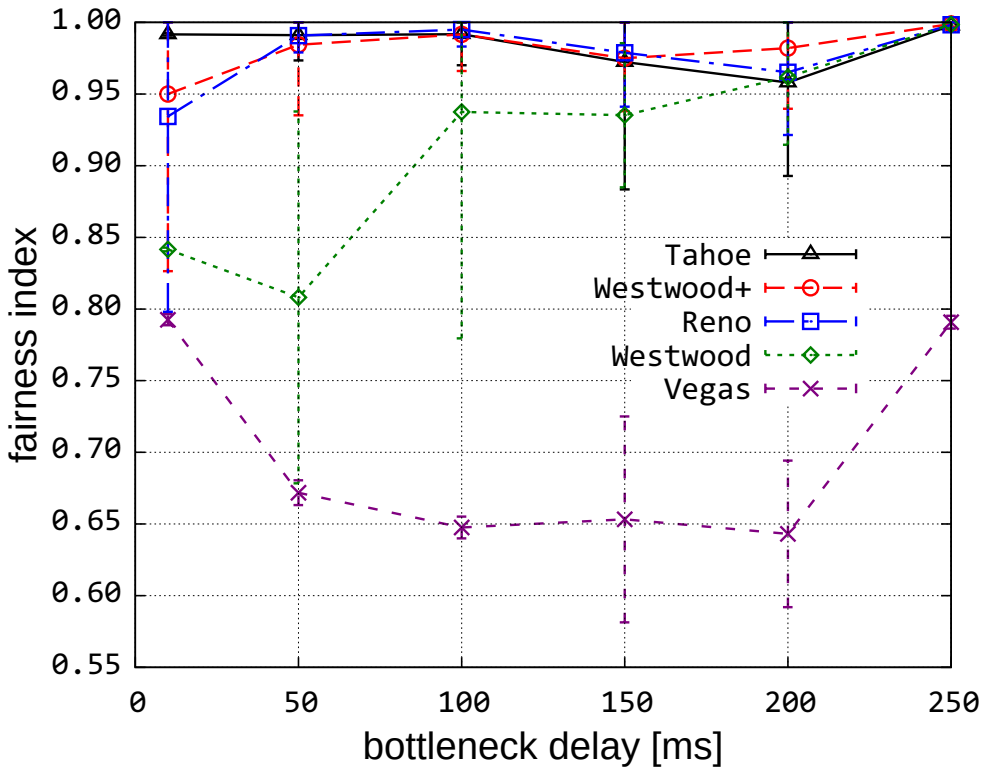

Figure 4.10. Friendliness vs. increasing bottleneck delay

4.2.5 Intra-Protocol Fairness

We measure the intra-protocol fairness of TCP variants by simulating two flows of the same TCP over the bottleneck link. All the protocols achieve very high

fairness regardless of the increasing in the bottleneck delay. Again, at a very low delay, Westwood suffers due to the small queue size and the flow's high sending rate.

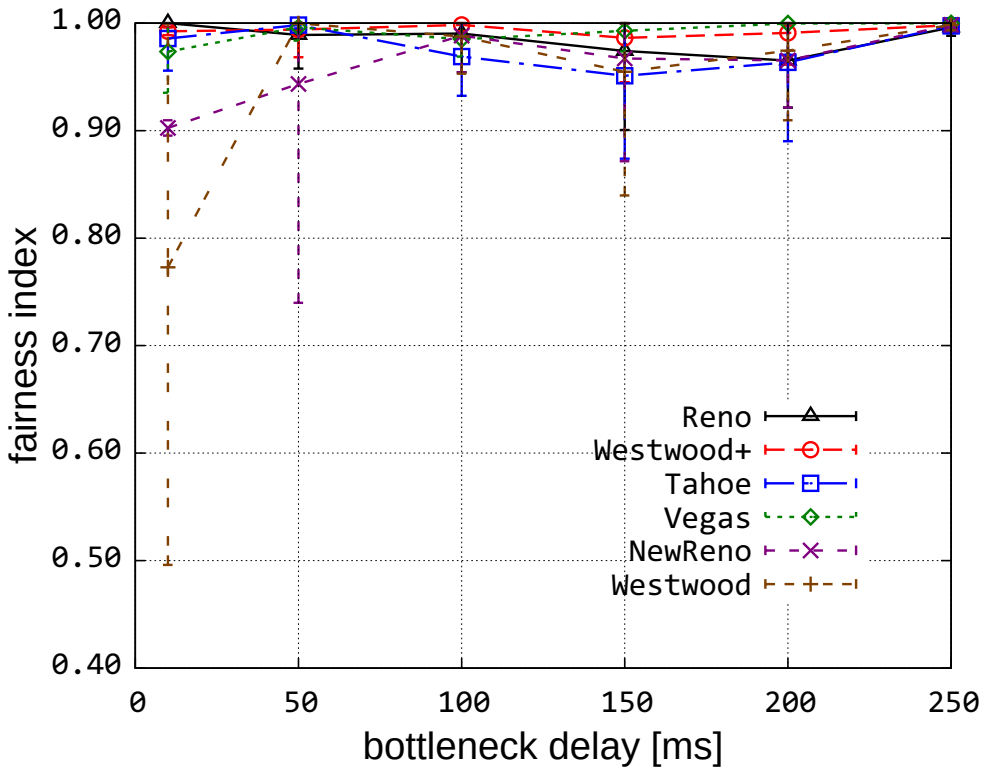

Figure 4.11. Intraprotocol fairness vs. increasing bottleneck delay

4.2.6 Summary

In summary, based on the results we obtain in our simulations, as long as the PER does not exceed 10^{-4}, Westwood+ performs much better than Westwood. In the high bandwidth and high delay scenarios, Westwood exhibits a very unstable behavior due to its use of ACK interarrival as the bandwidth sampling interval although to fully understand this instability, we need further investigations. The Vegas mechanism in trying to predict congestion before it actually occurs prevents

Vegas from fully utilize the available bandwidth and becomes uncompetitive with the existence of other TCP flows even though it has the best performance in the RTT fairness case. The additive increase, multiplicative decrease (AIMD) method in Tahoe, Reno, NewReno, or even in SACK degrades their performance in network environments with lossy links. For these protocols to be able to utilize the bandwidth, the PER has to be very small, which is normally impractical. The poor performance of Tahoe indicates the importance of the Fast Recovery mechanism, which allows the sender to go back to the congestion avoidance phase after a loss instead of starting over from the slow start phase, in improving TCP performance. However, these protocols are fair and friendly when they have to compete with other protocols, except Vegas. Among all the protocols studied in this thesis, we would suggest the use of Westwood+ in either a wired environment in which congestion is normally the main cause of packet drops or a wireless environment in which corruption-based losses occur more often.

Chapter 5

Conclusions and Future Work

This chapter concludes the thesis with some remarks and highlights based on the performance comparison results in the previous Chapter and gives directions for future work.

5.1 Conclusions

The thesis provides a baseline performance comparison work on different reliable transport-layer mechanisms to prepare the knowledge for future development of our resilient transport protocol ResTP. The results on the different mechanisms suggest that SNACK, the hybrid version of SACK and NAK, has the best performance due to its ability to inform more information about the receiver buffer than the other protocols. For congestion control algorithms, we simulate the standard TCP variants including Tahoe, Reno, NewReno, Vegas, Westwood, and Westwood+. While Westwood is known for improving TCP in facing corruption-based losses, its bandwidth estimation mechanism and its ACK-based sampling method result in some fairness issues. Vegas, on the other hand, highlights the difference

between a loss-based and a delay-based congestion control algorithm.

5.2 Future work

For future work, we would like to consider introducing background traffic into our congestion control testing to make the scenarios more realistic. Synchronization and phase effects could be studied at a deeper level. The impacts of different queue types other than drop tail on the protocols are also our interests. With the existing high-speed protocols currently available in the kernel, we also consider performing a more comprehensive analysis of the congestion control algorithms using more metrics.

References

[1] James P.G. Sterbenz. End-to-End Transport. http://www.ittc.ku.edu/ jpgs/courses/nets/lecture-transport-nets-display.pdf, February 2011.

[2] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), 1996.

[3] R. Fox. TCP big window and NAK options. RFC 1106, June 1989.

[4] CCSDS-The Consultative Committee for Space Data Systems. Space Communications Protocol Specification (SCPS)-Transport Protocol (SCPS-TP). http://public.ccsds.org/publications/archive/714x0b2.pdf, October 2006.

[5] Jim Kurose and Keith Ross. *Computer Networking: A Top-Down Approach.* Pearson Addison Wesley, 5th edition, 2010.

[6] Sami Iren, Paul D. Amer, and Phillip T. Conrad. The transport layer: tutorial and survey. *ACM Computing Surveys*, 31(4):360–404, December 1999.

[7] J. Postel. Transmission Control Protocol. RFC 793 (Standard), 1981. Updated by RFCs 1122, 3168.

[8] The ns-3 network simulator. http://www.nsnam.org, July 2009.

[9] Justin P. Rohrer, Erik Perrins, and James P.G. Sterbenz. End-to-End Disruption-Tolerant Transport Protocol Issues and Design for Airborne Telemetry Networks. In *Proceedings of the International Telemetering Conference*, San Diego, CA, October 27–30 2008.

[10] Justin P. Rohrer, Abdul Jabbar, Egemen K. Çetinkaya, Erik Perrins, and James P.G. Sterbenz. Highly-Dynamic Cross-Layered Aeronautical Network Architecture. *Aerospace and Electronic Systems, IEEE Transactions on*, 47(4):2742 –2765, October 2011.

[11] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm. RFC 3517 (Standard), 2003.

[12] V. Jacobson and R.T. Braden. TCP extensions for long-delay paths. RFC 1072, 1988. Obsoleted by RFCs 1323, 2018, 6247.

[13] L. Eggert and Nokia. Moving the Undeployed TCP Extensions RFC 1072, RFC 1106, RFC 1110, RFC 1145, RFC 1146, RFC 1379, RFC 1644, and RFC 1693 to Historic Status. RFC 6247 (Informational), May 2011.

[14] Ruhai Wang. A novel acknowledgment scheme for space Internet. In *Vehicular Technology Conference, 2004. VTC2004-Fall. 2004 IEEE 60th*, volume 6, pages 4056–4060 Vol. 6, 2004.

[15] Robert C. Durst, Gregory J. Miller, and Eric J. Travis. TCP extensions for space communications. *Wireless Networks*, 3(5):389–403, October 1997.

[16] Andrea Baiocchi, Angelo P Castellani, and Francesco Vacirca. YeAH-TCP: yet another highspeed TCP. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.

[17] Van Jacobson. Congestion Avoidance and Control. In *Symposium proceedings on Communications architectures and protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

[18] Van Jacobson. Modified TCP congestion avoidance algorithm. ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt, April 1990.

[19] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Obsoleted by RFC 5681, updated by RFC 3390.

[20] Janey C. Hoe. Start-up dynamics of TCP's congestion control and avoidance schemes. Master's thesis, University of California at Berkeley, USA, 1995.

[21] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582 (Experimental), April 1999. Obsoleted by RFC 3782.

[22] S. Mascolo, C. Casetti, M. Gerla, M.Y. Sanadidi, and R. Wang. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *MOBICOM 2001*, pages 287–297. ACM.

[23] S. Mascolo, L.A. Grieco, R. Ferorelli, P. Camarda, and G. Piscitelli. Performance evaluation of Westwood+ TCP congestion control. *Performance Evaluation*, 55(1-2):93–111, January 2004.

[24] Tom Kelly. Scalable TCP: improving performance in highspeed wide area networks. *SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, April 2003.

[25] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), December 2003.

[26] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2514–2524, 2004.

[27] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[28] Raj Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *SIGCOMM Comput. Commun. Rev.*, 19(5):56–71, October 1989.

[29] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, 1994.

[30] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, December 2006.

[31] Ryan King, Richard Baraniuk, and Rudolf Riedi. TCP-Africa: an adaptive and fair rapid increase rule for scalable TCP. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1838–1848, 2005.

[32] Kun Tan, Jingmin Song, Qian Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, 2006.

[33] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):89–102, August 2002.

[34] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *SIGCOMM Computer Communication Review*, 26(3):5–21, July 1996.

[35] Jeffrey C. Mogul. Observing TCP dynamics in real networks. *SIGCOMM CCR*, 22(4):305–317, October 1992.

[36] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[37] Ming Zhang, Brad Karp, Sally Floyd, and Larry Peterson. RR-TCP: a reordering-robust TCP with DSACK. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings*, pages 95–106, 2003.

[38] Rajkumar Kettimuthu and William Allcock. Improved Selective Acknowledgment Scheme for TCP. Technical report, Argonne National Laboratory, Globus Alliance, June 2004.

[39] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007.

[40] Preethi Natarajan, Nasif Ekiz, Ertugrul Yilmaz, Paul D. Amer, Janardhan Iyengar, and Randall Stewart. Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP. In *IEEE International Conference on Network Protocols, 2008. ICNP 2008*, pages 187–196, 2008.

[41] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), 2011.

[42] Fan Yang and Paul Amer. Non-renegable Selective Acknowledgments (NR-SACKs) for MPTCP. In *27th International Conference on Advanced Information Networking and Applications Workshops (WAINA), 2013*, pages 1113–1118, 2013.

[43] Abhay Chrungoo, Vishu Gupta, Huzur Saran, and Rajeev Shorey. TCP k-SACK: a simple protocol to improve performance over lossy links. In *Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE*, volume 3, pages 1713–1717 vol.3, 2001.

[44] Matthew Mathis and Jamshid Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. *SIGCOMM Computer Communication Review*, 26(4):281–291, August 1996.

[45] Matt Mathis and Jamshid Mahdavi. TCP Rate-Having with Bounding Parameters. Technical report, Pittsburgh Supercomputer Center, October 1996.

[46] K. N. Srijith, LiUykutty Jacob, and A. L. Ananda. TCP Vegas-A: solving the fairness and rerouting issues of TCP Vegas. In *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, pages 309–316, 2003.

[47] Amir Maor and Yishay Mansour. AdaVegas: adaptive control for TCP Vegas. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 7, pages 3647–3651 vol.7, 2003.

[48] Joel Sing and Ben Soh. TCP New Vegas: Improving the Performance of TCP Vegas Over High Latency Links. In *Network Computing and Applications, Fourth IEEE International Symposium on*, pages 73–82, 2005.

[49] Hong fei Liu, Li jun Li, Zu yuan Yang, and Xi yue Huang. TCP Vegas_M: A Novel TCP Algorithm in Mobile Ad hoc Network. In *6th International Conference on ITS Telecommunications Proceedings, 2006*, pages 629–633, 2006.

[50] Cheng-Yuan Ho, Yi-Cheng Chan, and Yaw-Chung Chen. Gallop-Vegas: An enhanced slow-start mechanism for TCP Vegas. *Journal of Communications and Networks*, 8(3):351–359, 2006.

[51] C. Y Ho and Y.-C. Chen. Snug-Vegas and Snug-Reno: efficient mechanisms for performance improvement of TCP over heterogeneous networks. *Communications, IEE Proceedings-*, 153(2):169–176, 2006.

[52] Lianghui Ding, Xinbing Wang, Youyun Xu, Wenjun Zhang, and Wen Chen. Vegas-W: An Enhanced TCP-Vegas for Wireless Ad Hoc Networks. In *Communications, 2008. ICC '08. IEEE International Conference on*, pages 2383–2387, 2008.

[53] Wei Zhou, Wei Xing, Yongchao Wang, and Jianwei Zhang. TCP Vegas-V: Improving the performance of TCP Vegas. In *Automatic Control and Artificial Intelligence (ACAI 2012), International Conference on*, pages 2034–2039, 2012.

[54] Thomas Bonald. Comparison of {TCP} reno and {TCP} vegas: efficiency and fairness. *Performance Evaluation*, 36â37(0):307 – 332, 1999.

[55] Go Hasegawa, Kenji Kurata, and Masayuki Murata. Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet. In *Network Protocols, 2000. Proceedings. 2000 International Conference on*, pages 177–186, 2000.

[56] Catherine Boutremans and Jean-Yves Le Boudec. A note on the fairness of TCP Vegas. In *Broadband Communications, 2000. Proceedings. 2000 International Zurich Seminar on*, pages 163–170, 2000.

[57] Steven H. Low, Larry L. Peterson, and Limin Wang. Understanding TCP Vegas: a duality model. *J. ACM*, 49(2):207–235, March 2002.

[58] Rung-Shiang Cheng and Hui-Tang Lin. TCP Selective Negative Acknowledgment over IEEE 802.11 Wireless Networks. In *International conference on Networking and Services, 2006. ICNS '06*, pages 98–98, 2006.

[59] Ru H. Wang, Satinderbir Singh, Sreelakshmi Bonasu, and Guangbin Fan. An experimental evaluation of a novel acknowledgment scheme over GEO-satellite links. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 3, pages 1491–1496 Vol. 3, 2005.

[60] Anurag Kumar. Comparative performance analysis of versions of TCP in a local network with a lossy link. *IEEE/ACM Trans. Netw.*, 6(4):485–498, August 1998.

[61] Michele Zorzi, A. Chockalingam, and Ramesh R. Rao. Throughput analysis of TCP on channels with memory. *IEEE Journal on Selected Areas in Communications*, 18(7):1289–1300, 2000.

[62] Haewon Lee, Soo-Hyeong Lee, and Yanghee Choi. The influence of the large bandwidth-delay product on TCP Reno, NewReno, and SACK. In *Information Networking, 2001. Proceedings. 15th International Conference on*, pages 327–334, 2001.

[63] Luigi A. Grieco and Saverio Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *SIGCOMM Comput. Commun. Rev.*, 34(2):25–38, April 2004.

[64] Alaa Seddik-Ghaleb, Yacine Ghamri-Doudane, and Sidi-Mohammed Senouci. A performance study of TCP variants in terms of energy consumption and average goodput within a static ad hoc environment. In *Proceedings of the 2006 international conference on Wireless communications and mobile computing*, IWCMC '06, pages 503–508, New York, NY, USA, 2006. ACM.

[65] Siddharth Gangadhar, Truc Anh N. Nguyen, Greeshma Umapathi, and James P.G. Sterbenz. TCP Westwood Protocol Implementation in ns-3. In *Proceedings of the ICST SIMUTools Workshop on ns-3 (WNS3)*, Cannes, France, March 2013.

[66] The ns-3 Network Simulator Doxygen Documentation. http://www.nsnam.org/doxygen/group_tcp.html, July 2012.

[67] S. Floyd. Congestion Control Principles. RFC 2914 (Best Current Practice), September 2000.

[68] Linux Kernel Organization. The Linux Kernel Archives. https://www.kernel.org, 2013.

[69] Larry Peterson. The x-kernel Protocol Framework. http://www.cs.arizona.edu/projects/xkernel/, 2013.

[70] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.